# CURSO
# DE INICIACIÓN
# A LA PROGRAMACIÓN
# DE VIDEOJUEGOS
# CON EL LENGUAJE
# BENNU v1.0
## (en Windows y GNU/Linux)

**Oscar Torrente Artero**

# Agradecimientos

Quiero agradecer este texto a todas las personas que, gracias al empeño y las ganas que demuestran y transmiten diariamente, tiran para adelante este proyecto tan fantástico que es Bennu, y a toda la gente que participa en el foro aportando toda la ayuda que está en su mano. Gran parte de los códigos de ejemplo presentes en esta guía han sido extraidos de allí. Gracias por todo.

# A quién está dirigido este texto

Este curso está pensado para usuarios con conocimientos medios de informática (p. ej: que saben qué es la memoria RAM o la extensión de un archivo, que saben crear un acceso directo, que han utilizado alguna vez algún gestor de paquetes ,o que han oido hablar alguna vez de la linea de comandos...), ya sea en Windows como en cualquier distribución GNU/Linux, pero que no hayan programado nunca, y que quieran aprender utilizando para ello el lenguaje Bennu. Por lo tanto, está pensado para gente que se quiera iniciar en el mundo de la programación a través de la excusa de la creación videojuegos, objetivo éste apasionante y realmente reconfortante.Y finalmente, está pensado para gente con ilusión para ilusionar (valga la redundancia) a los demás creando mundos y experiencias mágicas y únicas.

El objetivo de este texto es ser para el futuro programador una introducción elemental en el fascinante mundo de la algoritmia y del desarrollo de sotware mediante un lenguaje potente y flexible como Bennu, asumiendo que el lector parte de un nivel de conocimiento escaso en lo que se refiere a los conceptos básicos de programación. Por tanto, la intención de este libro es enseñar a programar a aquellos que lo intentan por primera vez y facilitar la adquisición de los procedimientos básicos necesarios para el desempeño en esta actividad, utilizando como excusa los videojuegos y como medio el lenguaje Bennu. Por eso no se han incluido temas avanzados que sobrepasarían el objetivo inicial planteado.

Así mismo, este curso no es una referencia o compendio exhaustivo de las funciones y comandos que Bennu aporta. El lector experimentado notará que en las páginas siguientes faltarán por mencionar y explicar multitud de funciones interesantes ( en concreto, las de manejo del CD y del joystick, las funciones de encriptación,, las de control de animaciones FLI/FLC, todas las funciones relacionadas con la manipulación de paletas de color y todas las funciones pertenecientes a módulos no oficiales como la Fsock ó la Bennu3D, etc).No se ha pretendido nunca que este texto albergara toda la información sobre Bennu, y sólo se ha incidido en aquellos aspectos del lenguaje que han sido considerados más relevantes, importantes o útiles en el día a día de un programador de este lenguaje.

# Cómo leer este texto

La presente guía se ha escrito teniendo en mente varios aspectos. Se ha procurado en la medida de lo posible escribir un manual que sea **autocontenido** y **progresivo.** Es decir, que no sea necesario recurrir a fuentes de información externas para comprender todo lo que se explica, sino que el propio texto sea autoexplicativo en sí mismo, y que toda la información expuesta sea mostrada de forma ordenada y graduada, sin introducir conceptos o procedimientos sin haberlos explicado con anterioridad.

La metodología utilizada en este texto se basa fundamentalmente en la exposición y explicación pormenorizada de multitud de ejemplos de código **cortos y concisos**: se ha intentado evitar códigos largos y complejos, que aunque interesantes y vistosos, pueden distraer y desorientar al lector al ser demasiado inabarcables.

La estructura de los capítulos es la siguiente: podemos considerar una primera parte del manual integrada por los capítulos 1,2,3,4 y 5 que consiste en la exposición progresiva de la instalación y configuración del entorno Bennu y de toda la sintaxis y elementos gramaticales del propio lenguaje Bennu (variables, estructuras de control, estructuras de datos,manejo de gráficos, etc) a modo de un típico manual de programación al uso. Posteriormente, podemos considerar una segunda parte del manual, integrada por los capítulos 6 y 7, que consiste en la aplicación práctica de los conocimientos aprendidos en los capítulos anteriores a tutoriales de videojuegos concretos. Seguidamente, el capítulo 8 recoge una gran variedad de información sobre diferentes comandos del lenguaje Bennu, logrando un exhaustivo repaso de la mayoría de funcionalidades del lenguaje, obteniendo con su lectura un conocimiento ya bastante avanzado de las posibilidades que depara Bennu. El capítulo 9 vuelve a ser un tutorial al estilo de los capítulos 6 y 7, pero incluyendo gran parte de lo explicado en el capítulo 8. El capítulo 10 se compone de una miscelánea de aspectos del lenguaje Bennu que bien por su complejidad o por su especificidad no se han tratado anteriormente. El capítulo 11 reúne un conjunto de códigos variados (algunos más curiosos, otros más útiles, pero siempre cortos) como muestra de diferentes ideas que se pueden realizar con Bennu. Finalmente, se han incluido dos apéndices: el primero es una introducción a los conceptos básicos de la programación, especialmente recomendable para los lectores que no han programado nunca antes y ni siquiera saben lo que es un lenguaje de programación; el segundo es una introducción a los conceptos básicos de multimedia (video, audio, 3D, gráficos), imprescindibles de conocer para poder afrontar la programación de un videojuego (al fin y al cabo, una aplicación multimedia) con plenas garantías.

# INDICE

## CAPITULO 1: PRIMER CONTACTO CON BENNU



## CAPITULO 2: EMPEZANDO A PROGRAMAR CON BENNU

## CAPITULO 3: GRÁFICOS



## CAPITULO 4: DATOS Y PROCESOS



## CAPITULO 5: ESTRUCTURAS DE DATOS

## CAPITULO 6: TUTORIALES BÁSICOS: EL JUEGO DEL LABERINTO Y UN PING-PONG



## CAPITULO 7: TUTORIAL PARA UN MATAMARCIANOS



## CAPITULO 8: ÓRDENES Y ESTRUCTURAS BÁSICAS DE BENNU

## CAPITULO 9:TUTORIAL PARA UN RPG ELEMENTAL



## CAPITULO 10:ÓRDENES Y ESTRUCTURAS AVANZADAS DE BENNU



## CAPITULO 11: CODIGOS Y TRUCOS VARIADOS

## APÉNDICE A: CONCEPTOS BÁSICOS DE PROGRAMACIÓN



## APÉNDICE B: CONCEPTOS BÁSICOS DE MULTIMEDIA

# CAPITULO 1

## Primer contacto con Bennu

### *¿Qué es Bennu? Propósito y características

Bennu es un sencillo lenguaje de programación libre y multiplataforma (existen versiones oficiales para sistemas Windows, GNU/Linux y OpenWiz) de tipo compilado/interpretado y que está específicamente diseñado para crear y ejecutar cualquier tipo de juego 2D. Tal como dice su página web oficial: http://www.bennugd.org , Bennu es un entorno de desarrollo de videojuegos de "alto nivel" que da especial énfasis a la portabilidad del código, (proporcionando al desarrollador la posibilidad de escribir su programa una sola vez pero pudiéndolo ejecutar en múltiples plataformas sin modificar absolutamente nada), y también a la modularidad, (existiendo múltiples librerías que extienden y complementan la funcionalidad básica de Bennu: librerías de red, librerías 3D, de renderizado de textos, de acceso a bases de datos y archivos XML, etc). Podemos decir, pues, que con Bennu tanto los programadores noveles como los avanzados encontrarán una inestimable herramienta para sus lograr hacer realidad sus creaciones.

El lenguaje está claramente orientado al manejo de gráficos 2D; es decir, que el propio lenguaje se preocupa de dibujar los gráficos en pantalla, ahorrándole al programador el trabajo que eso supone, que es mucho. Bennu incluye de serie un motor de renderizado 2D por software que convierte la programación de juegos en una tarea fácil pero potente. En general, cualquier juego que no emplee objetos 3D real es posible realizarlo con Bennu sin necesidad de ninguna librería extra. Más concretamente, algunas de sus características principales son:

*Dibujo rápido de sprites con rotado, escalado, animación y grados de transparencia.
*Detección de colisiones a nivel de píxel
*Procesos (programación multihilo)
*Entrada por teclado, ratón y joystick
*Modos de color de 8, 16 y 24/32 bits
*Soporte del formato gráfico Png, entre otros.
*Soporte de los formatos de sonido Wav sin compresión y Ogg Vorbis,además de módulos de música It,Mod y Xm
*Rutinas de scroll tipo "parallax"
*Múltiples regiones en pantalla con o sin scroll
*Recorrido de caminos inteligente
*Rutinas matemáticas y de ordenación
*Rutinas de acceso y manipulación de directorios y ficheros
*Consola de depuración
*Uso de temporizadores
*Comprobación de expresiones regulares y manejo de cadenas de caracteres complejas
*Manejo dinámico de memoria
*Soporte para Modo 7
*Soporte para el cifrado de recursos mediante criptografía simétrica
*Soporte de librerías externas (en Windows, *.dll; en GNU/Linux, *.so).
*Multiplataforma: funciona de forma oficial en todos los Windows (incluido Windows 7), en GNU/Linux sobre chips Intel y en la consola GP2X Wiz (http://www.gp2xwiz.com). Pero es factible portar Bennu a múltiples plataformas más, como por ejemplo MacOSX, PSP, Xbox, Wii y PlayStation, gracias a que Bennu está basado a más bajo nivel en la librería gráfica SDL (ver Apéndice A) , la cual es portable a todos estas plataformas mencionadas. De hecho, existe un port no oficial de Bennu para la consola Wii, disponible en http://code.google.com/p/bennugd-wii

Incluso programadores experimentados encontrarán en Bennu una herramienta útil. El lenguaje en sí soporta muchas de las características comúnmente encontradas en los lenguajes procedurales, tales como múltiples tipos de dato , punteros, tablas multidimensionales, estructuras, y las sentencias habituales de control de flujo. De hecho, Bennu se puede interpretar como un recubrimiento, una capa superior de la librería SDL, que encapsula, que oculta la dificultad intrínseca que tiene a la hora de programar con ella en C.

Los autores de Bennu creen en la filosofía de la libertad de uso del software y en la distribución del código fuente. El lenguaje Bennu, su compilador y su intérprete se distribuyen gratuitamente bajo los términos de la GNU General Public License, con lo que las fuentes están disponibles y eres libre de extenderlas o hacer tu propia versión, tal como se comenta en el Apéndice A. La distribución de Bennu bajo la licencia GNU GPL implica que, si realizas cambios en Bennu, estás obligado a distribuir el código fuente que hayas cambiado junto con tu versión, sin ningún coste, y bajo la misma licencia, permitiendo su redistribución. Eso sí, esto no afecta para nada a los juegos que puedas desarrollar gracias a Bennu. los juegos realizados con Bennu están libres de cualquier limitación y podrás ponerles la licencia que quieras, así como cualquier librería realizada por ti (si no deriva de terceros) que enlace con dichos juegos.

## *Historia de Bennu

En la década de los 90 el entonces estudiante Daniel Navarro Medrano creó como proyecto final de carrera una herramienta orientada a la creación de videojuegos de 32 bits bajo MS-DOS. El nuevo lenguaje, de nombre DIV Games Studio, combinaba características de C y Pascal con un entorno completo que permitía la creación y edición de todos los aspectos de los proyectos: programación, edición gráfica y sonora y un largo etc.

Fénix, inicialmente bajo el nombre DIVC y de naturaleza GNU y gratuita, apareció de la mano de Jose Luis Cebrián como una herramienta capaz de compilar y ejecutar esos juegos en GNU/Linux. El nombre fue cambiado en la versión 0.6 del compilador, que además introducía otras mejoras, como la aparición de un fichero intermedio entre el entorno de compilación y el entorno de ejecución. Ya no era necesario distribuir el código fuente de un juego para poder jugar a los juegos. La ventaja principal de esa práctica (similar en concepto a Java) era clara, compilar en una plataforma y ejecutar en muchas. En la versión 0.71 el proyecto quedó parado, lo que dio lugar a múltiples versiones derivadas que corregían fallos o añadían nuevas características.

La versión oficial de Fénix fue retomada por Slàinte en el año 2002, viejo conocido de la comunidad DIV por ser el webmaster de una de las páginas web más importantes para la comunidad, quien continuó el proyecto bajo el nombre de “Fénix-Proyecto 1.0” al que pronto se reincorporaría su creador y cuyo primer objetivo era limpiar el compilador de errores y estabilizarlo. Desde entonces el compilador ha sufrido numerosos cambios y mejoras, dejando de un lado la compatibilidad con el lenguaje DIV, el desarrollo del cual quedó paralizado hace tiempo en su versión 2 desde la quiebra de la empresa que lo comercializaba, Hammer Technologies. (De hecho, DIV2 sólo es compatible con Ms-Dos y Windows 95/98).

Tras un periodo de relativa inactividad en donde Fénix se había estancado en su versión 0.84/0.84a, a mediados del 2006, Juan alias "SplinterGU" retomó con fuerza el desarrollo de nuevas versiones del compilador/intérprete, incorporándole muchas mejoras en rendimiento y velocidad, añadiendo funciones nuevas y corrigiendo bugs crónicos, y modularizando por completo todo el entorno (una demanda mucho tiempo reclamada), hasta llegar a la versión 0.92a. No obstante, ciertas desaveniencias con determinados miembros de la comunidad hicieron que Juan se decidiera por abandonar el desarrollo oficial de Fénix y crear un fork (una variante independiente) a partir del código de Fénix llamado Bennu. Bennu es el nombre de un dios egipcio, cuya apariencia física es la de una garza real llevando sobre la cabeza la corona del Alto Egipto. Este dios es el de la regeneración (cómo Fénix), debido al carácter de ave migratoria que reaparece, que renace, que se renueva periódicamente.

A partir de 2007, poco a poco se ha ido construyendo una nueva comunidad alrededor de este nuevo lenguaje renacido. Hoy en día el foro de Bennu (http://forum.bennugd.org) es un lugar muy activo lleno de gente que intercambia información, trucos y experiencias sobre este lenguaje y que están encantados de echar una mano. Además, la comunidad ofrece muchos otros recursos para el programador: desde a la documentación online del lenguaje, en forma de wiki (http://wiki.bennugd.org) -donde se pretende recoger la descripción de todas las funciones, variables, características sintácticas y entresijos del lenguaje- hasta diferentes proyectos, librerías y aplicaciones software que facilitan mucho el día a día a la hora de programar, y las cuales iremos estudiando en este capítulo.

## *Carencias de Bennu v1.0

Hay algunas carencias de Bennu que a primera vista podrían sorprender al recién llegado. Por ejemplo, la inexistencia de un IDE oficial propio.

*Un IDE (Integrated Development Environment, ó "Entorno de Desarrollo Integrado") no es más que un programa que integra en un solo entorno el editor, el intérprete, el compilador, el depurador si lo hubiera, la ayuda del lenguaje, etc; de manera que todas las herramientas necesarias para el programador están accesibles inmediatamente dentro de ese entorno de forma coherente,cómoda y sobretodo, visual. Si no se trabaja dentro de un IDE, las diferentes herramientas –que en el caso de Bennu son de consola (es decir, ejecutables vía línea de comandos -la famosa "ventanita negra"-, y por tanto no visuales )- son para el desarrollador programas independientes, inconexos entre sí, con lo que la utilización conjunta de uno u otro es más pesada e incómoda, y no es visual).*

La razón de que no haya un IDE oficial que acompañe a las versiones oficiales del lenguaje es porque el compilador y el intérprete Bennu son programas multiplataforma, y su entorno oficial debería obligatoriamente funcionar en cualquier plataforma donde funciona Bennu. El mejor modo de conseguir esto sería programar dicho entorno mediante el propio Bennu, pero todavía no hay ningún resultado en este aspecto. No obstante, sí que existen diversos IDEs no oficiales, que podremos disfrutar sólo en alguna plataforma concreta (sólo en Windows, sólo en GNU/Linux, etc), los cuales serán comentados más adelante en este capítulo.

Otra carencia que puede sorprender es la inexistencia (aparente) de soporte 3D. "Aparente" porque sí que existen librerías (no oficiales) que permiten trabajar en 3D (como la Bennu3D ó la BeOgre, entre otras: ya las comentaremos en posteriores capítulos) con resultados más que satisfactorios. No obstante, Bennu ha sido diseñado de entrada para realizar juegos 2D y un motor 3D es un cambio fundamental que implica cambiar prácticamente todo el intérprete.

Otra carencia es la (aparente, otra vez) inexistencia de rutinas de red, para juegos on-line. "Aparente" porque sí que existen librerías (no oficiales) que permiten trabajar en red (como la Fsock ó la Net derivada de la SDL_net, entre otras: ya las comentaremos en posteriores capítulos). Sin embargo, este tipo de librerías requerirán que el programador haga la mayor parte del trabajo enviando y recibiendo paquetes de red , y controlando por su cuenta paquetes perdidos o recibidos fuera de secuencia y demás problemas habituales. Programar un juego en red es un arte complejo.

## *Instalación de Bennu en Windows

Para descargar Bennu, tendrás que ir a la sección de "Downloads" de la página oficial: http://www.bennugd.org . Allí verás que tienes varias opciones a elegir, pero la que ahora nos interesa seleccionar a nosotros es, obviamente, el instalador para Windows. Como curiosidad, es interesante saber que este instalador ha sido sido programado usando el propio lenguaje Bennu, con lo que queda demostrada la capacidad y versatilidad de dicho entorno para realizar múltiples y avanzadas tareas. Bien, una vez descargado este ejecutable (que ocupa tan sólo 2 MB), si lo pones en marcha verás cómo aparece una ventana que simula la apariencia de una especie de consola portátil, donde se pueden apreciar a la izquierda varios botones que realizan diferentes tareas y un panel central que lista las carpetas de nuestro disco.

Lo que tienes que hacer para instalar Bennu es elegir la carpeta de tu disco duro donde copiarás todos los archivos que lo componen. Para ello simplemente has de clicar una vez sobre la carpeta elegida (verás que se marca de color azul). Puedes asegurarte de que efectivamente Bennu se instalará bajo la carpeta que has dicho si observas el mensaje deslizante que aparece en la parte inferior de la ventana, el cual muestra precisamente la ruta de instalación elegida. Si lo deseas,puedes moverte por el árbol de carpetas hacia adelante y hacia atrás sin más que clicar dos veces sobre la carpeta a la que quieres entrar (recuerda, por si no lo sabías, que para volver a la carpeta anterior tendrías que clicar en la carpeta especial "..."). También existe la posibilidad de crear una nueva carpeta allí donde estés en ese momento, pulsando el botón adecuado; si quieres cambiar el nombre de esta nueva carpeta lo puedes hacer clicando con el botón derecho sobre ella (verás que queda marcada en rojo) y finalizando el cambio pulsando la tecla "Enter".

Una vez elegida la carpeta base de la instalación de Bennu, tan sólo queda pulsar en el botón de "Install".

El proceso de instalación de Bennu en realidad es muy sencillo. Lo único que hace son básicamente dos cosas: 1º) copiar todos los ficheros que componen Bennu a la carpeta elegida, y 2º) incluir las rutas necesarias en el PATH del sistema. Y ya está. Pero expliquémoslo más detenidamente.

Respecto el primer punto, suponiendo que por ejemplo la carpeta que hemos elegido para la instalación de Bennu hayamos dicho que es C:\BennuGD, el instalador copiará allí cuatro subcarpetas:

| | |
|---|---|
| **C:\BennuGD\bin** | Contiene (tal como su nombre indica: "bin" de "binario") los ejecutables que utilizará el programador: el compilador (*bgdc.exe*), el intérprete (*bgdi.exe*) -ver Apéndice A- y otra utilidad de consola de la que hablaremos más tarde: *moddesc.exe*. |
| **C:\BennuGD\modules** | Contiene todos los módulos oficiales que componen el entorno Bennu y que están a disposición del programador. Un módulo es una librería separada del resto que contiene cierta funcionalidad específica muy concreta. Podemos encontrar el módulo Bennu para el manejo del teclado, el módulo Bennu que se encarga del dibujado de sprites, el módulo Bennu encargado de la gestión del sonido, etc. Dicho de otra forma, los comandos del lenguaje Bennu están organizados y separados por módulos, de manera que un comando concreto está "ubicado" dentro de un módulo concreto. Esto implica que para hacer uso de un comando particular, deberemos incluir el módulo que lo contiene, porque de lo contrario Bennu no sabrá qué significa ese comando.<br><br>Esta manera de separar la funcionalidad de Bennu (de "modularizarlo") tiene varias ventajas:<br>-Es más fácil solucionar errores y ampliar funcionalidad del propio Bennu si se mantiene una estructura separada y lo más independiente posible entre los diferentes componentes, ya que así se puede localizar mejor la fuente de los posibles errores y se pueden mejorar de manera mucho más sencilla las diferentes características pendientes.<br>-Permite al desarrollador de Bennu elegir exactamente qué módulos quiere incluir en su proyecto y qué módulos no les es necesario. Con esto se consigue que los proyectos sean mucho más ligeros y optimizados porque sólo incluyen exclusivamente los módulos que aportan los comandos que han sido utilizados en el código.<br>-Permite al desarrollador de Bennu elegir entre varios módulos disponibles para realizar una misma acción. La estructura modular permite que, por ejemplo, si hubiera la posibilidad de elegir entre dos módulos de gestión de sonido, al ser completamente independientes del resto de módulos del entorno, se podría elegir el que más conviniera sin alterar para nada el resto de módulos a utilizar, permitiendo así una mayor libertad para incluir (o excluir) diferentes componentes a nuestro proyecto. |
| **C:\BennuGD\libs** | Los módulos para poder funcionar correctamente necesitan de la existencia de las librerías incluidas en esta carpeta, las cuales el programador de Bennu nunca utilizará directamente nunca, pero que son imprescindibles para el entorno Bennu . Estas librerías son internas de Bennu, desarrolladas por y para Bennu. De hecho, muchos módulos distintos comparten código alojado en estas librerías, de manera que una misma rutina utilizada por varios módulos aparece centralizada en una sola librería, de la cual éstos, por lo tanto, dependen. |
| **C:\BennuGD\externals** | Tal como se comenta en el Apéndice A, el propio Bennu está escrito en C más SDL; es decir, se basa en la librería SDL, por lo que depende de ella para poder funcionar correctamente. Asimismo, también depende de unas pocas librerías más (SDL_Mixer, LibPng, Zlib, LigOgg y LibVorbis, etc). Todas estas librerías externas ("externas" porque no han sido desarrolladas por el equipo de Bennu, son librerías de terceros de las cuales Bennu depende debido a su estructura interna) son las que aparecen en esta carpeta. Al igual que con la carpeta "libs", el desarrollador de Bennu no ha de hacer nada con el contenido de esta carpeta. Esta carpeta sólo se genera en la instalación de Bennu en Windows, ya que en GNU/Linux las dependencias externas han de haber sido resueltas mediante la instalación de los paquetes pertinentes, de forma previa e independiente a la ejecución del instalador. |

Respecto el segundo punto, incluir una ruta en el PATH del sistema significa que se le dice al sistema operativo (Windows, en este caso) en qué carpetas se han instalado los diferentes componentes de Bennu, para que así Windows pueda encontrar sin problemas todos los ficheros que forman Bennu, sin que haya errores precisamente por no encontrar un determinado módulo o una librería concreta. Puedes comprobar que en el PATH se incluyen las cuatro rutas de la tabla anterior si vas, por ejemplo,al componente "Sistema" del Panel de Control de Windows, en concreto en

la pestaña “Opciones avanzadas”, botón “Variables de entorno”: verás en el apartado “Variables del sistema” una llamada Path. Si pulsas en el botón de Modificar, verás su valor actual, el cual estará compuesto por una serie de rutas (las que precisamente están incluidas en el PATH), entre las cuales tendrían que estar (si no están, malo; habría que añadirlas) las carpetas “bin”, “modules”, “libs” y “externals” copiadas bajo la carpeta de instalación de Bennu.

## *Instalación de Bennu en GNU/Linux

A diferencia del instalador de Bennu para Windows, el instalador de Bennu para GNU/Linux no incorpora de serie las librerías de las cuales depende (las “externals” de la tabla que comentamos en el apartado anterior), por lo que antes de proceder a cualquier instalación de Bennu, hay que procurar tener resueltas previamente todas estas dependencias. SI NO HACEMOS ESTO, A PESAR DE QUE EL INSTALADOR FUNCIONE CORRECTAMENTE, BENNU NO FUNCIONARÁ.

Concretamente, las dependencias de Bennu son las siguientes (ver Apéndice A para más información):

| | |
|---|---|
| **SDL** (http://www.libsdl.org) | Esta librería no depende de ninguna otra explícitamente. Claro está, no obstante, que al ser una librería gráfica, requerirá algún sistema de dibujado en pantalla previamente configurado, tal como un servidor X (http://www.x.org, presente de serie en la inmensa mayoría de sistemas PC) ó, en dispositivos empotrados, un servidor DirectFB mínimo (http://www.directfb.org). |
| **SDL_Mixer** ((http://www.libsdl.org/projects/SDL_mixer) | Esta librería a su vez tiene una serie de dependencias que se han de cumplir, como son las librerías LibOgg, LibVorbis y LibVorbisfile (http://www.xiph.org/downloads) para la reprodución de audio Ogg, LibSmpeg (http://www.icculus.org/smpeg) para la reproducción de audio Mp3 (aunque Bennu no tiene soporte para este formato) y LibMikMod (http://mikmod.raphnet.net) para la reproducción de módulos de sonido. |
| **LibPng** (http://www.libpng.org) | Esta librería a su vez tiene una dependencia que se ha de cumplir, que es la librería **Zlib** (http://zlib.net) , la cual se encarga de comprimir y descomprimir datos al vuelo, y es usada por Bennu en otros ámbitos también, fuera del tratamiento de imágenes (así que se podría considerar como otra dependencia más independiente). |

Deberemos, pues, antes de nada, ejecutar nuestro gestor de paquetes preferido y buscar y seleccionar los paquetes anteriores. Posiblemente sus nombres puedan ser ligeramente dispares a los mostrados aquí (cada distribución tiene su propio convenio de nomenclatura) e incluso puede que dependiendo de la distribución la versión de algunas dependencias varíe; por eso recomiendo leer la descripción de cada paquete que muestre el gestor de instalación antes de proceder. A pesar de esta pequeña dificultad, es seguro que todas estas librerías se encuentran muy fácilmente en los repositorios oficiales de las distribuciones más importantes, porque son librerías muy básicas y extendidas y son utilizadas por multitud de software. En cualquier caso, para instrucciones más detalladas sobre este tema, puedes consultar la wiki.

En el caso, por ejemplo, de Ubuntu, en el momento de la redacción de este manual, si quisiéramos instalar las librerías anteriores por el intérprete de comandos, sería así (es preferible respetar el orden descrito).

Para instalar la librería SDL:

```
sudo aptitude install libsdl1.2debian-all
```

Para instalar la librería SDL_Mixer (automáticamente se instalarán como dependencias las librerías: libogg0,

libvorbis0a, libvorbisfile3, libsmpeg0 y libmikmod2):

```
sudo aptitude install libsdl-mixer1.2
```

Para instalar la librería Zlib:

```
sudo aptitude install zlib1g
```

Para instalar la librería LibPng:

```
sudo aptitude install libpng3
```

Una vez instaladas las dependencias necesarias, ya podemos proceder a la instalación de Bennu propiamente.Para descargar Bennu, tendrás que ir a la sección de "Downloads" de la página oficial: http://www.bennugd.org . Allí verás que tienes varias opciones a elegir, pero la que ahora nos interesa seleccionar a nosotros es, obviamente, el instalador para GNU/Linux. Si lo descargas e intentas por curiosidad abrir su contenido con un editor de texto plano, verás que no es más que un Bash shell script, seguido al final por un conjunto de datos binarios que son precisamente todos los ficheros que forman Bennu comprimidos y compactados dentro del propio shell script. A diferencia del instalador de Windows, que estaba hecho en Bennu y era visual, éste no lo es: lo tendremos que ejecutar desde un intérprete de comandos. Así:

Primero hacemos que el shell script descargado tenga permisos de ejecución, para que pueda ejecutarse, precisamente:

```
chmod 755 bgd-1.0.0-linux-installer.sh
```

Y seguidamente lo ejecutamos (debemos hacerlo como usuario "root",así que escribimos el comando sudo al principio):

```
sudo bash ./bgd-1.0.0-linux-installer.sh
```

Importante no olvidarse del "./" justo antes del nombre del shell script.

Una vez escrita la línea anterior, automáticamente comenzará a ejecutarse el shell script, procediendo a la instalación del entorno Bennu. Podrás comprobar que, a diferencia del instalador de Windows, no pregunta en qué carpeta se desean copiar los ficheros que componen Bennu: esto es así porque el instalador de Linux siempre copiará los ficheros a una misma carpeta predefinida, sin posibilidad de elegir.

El mismo shell script que funciona de instalador también puede utilizarse para desinstalar completamente el entorno Bennu (esta opción en Windows no existe). Simplemente hay que invocarlo pasándole el parámetro adecuado:

```
sudo bash ./bgd-1.0.0-linux-installer.sh --remove
```

La línea anterior borrará por completo todo rastro de instalación anterior de Bennu (realizada por el mismo shell script).

Si deseas saber qué es lo que hace exactamente el shell script de (des)instalación de Bennu y conoces mínimamente los comandos básicos de cualquier shell de GNU/Linux, puedes intentar leer el código del instalador y ver cuál es el proceso que ha seguido. Básicamente, persigue los mismos objetivos que el instalador de Windows: copiar los ficheros que componen el entorno Bennu a una serie de carpetas concretas, y registrar su ubicación en el sistema de manera que el sistema operativo sepa dónde encontrar los distintos ficheros cada vez que requiera su uso.

Más concretamente, el compilador (*bgdc*) y el intérprete (*bgdi*) de Bennu son copiados en la carpeta /usr/bin y las librerías y los módulos son copiados a las carpetas /usr/lib/bgd/lib y /usr/lib/bgd/module, respectivamente, creándose en /etc/ld.so.conf.d/bgd.conf un registro de dichas librerías para que (tras ejecutar /sbin/ldconfig) el sistema las reconozca y las pueda utilizar. Remito a la tabla del apartado anterior para conocer qué función tienen cada uno de estos componentes dentro de Bennu (compilador, intérprete, módulos y librerías).

Si por alguna razón deseas obtener un simple paquete tgz que incluya todos los ficheros que componen el entorno Bennu sin más, lo puedes obtener a partir del propio instalador, ejecutando en el terminal el siguiente comando:

```
tail -n +61 bgd-1.0.0-linux-installer.sh > bennu.tgz
```

donde el número 61 es el valor de la variable “data_start” definida al principio del shell script instalador (este valor podría cambiar en futuras versiones de éste) y “bennu.tgz” es el paquete generado.

## *Instalación de Bennu a partir del código fuente (nivel avanzado)

Desde la misma página de descargas de la web oficial de Bennu se puede conseguir el código fuente más reciente de Bennu: es el paquete tar.gz (por si no lo sabes, estos paquetes son similares a los zip o los rar) llamado, muy apropiadamente, “TAR with latest source”.

Obtener el código fuente nos puede interesar para dos cosas: o bien para observar el funcionamiento interno de Bennu y mejorar alguna de sus características (aspecto éste muy avanzado y que requiere altos conocimientos de lenguaje C y de la librería SDL), o bien para (sin obligación de hacer ninguna modificación en el código) compilar Bennu a nuestro gusto y obtener nuestra copia del entorno totalmente personalizada, es decir, “a la carta”:con opciones (des)habilitadas bajo demanda, con (ex)inclusión de funcionalidades concretas, con optimizaciones para nuestra arquitectura hardware concreta, con instalación en rutas no estándar, etc, lista para instalar. Para más información, consulta el Apéndice A.

No obstante, hay que indicar que esta manera de proceder no debería ser la estándar: para un usuario normal y corriente, ya que los instaladores comentados en los apartados anteriores son las herramientas recomendadas para realizar una instalación limpia, rápida y fácil: compilar Bennu para instalarlo se puede considerar sobretodo un ejercicio pedagógico avanzado para aprender sobretodo lenguaje C y las herramientas de compilación y sistema asociadas. Sólo habría un motivo realmente razonable para preferir la compilación a la instalación estándar, que es en el posible supuesto que se haya incorporado al código fuente alguna mejora o corrección que no se haya trasladado todavía al instalador oficial y que haya por tanto un desfase de versiones, pero esta circunstancia, si se llegara a producirse, sería muy puntual y esporádica.

Para poder compilar Bennu, obviamente necesitarás primero de todo un compilador de C. En GNU/Linux existe un programa que es EL compilador de C por excelencia, base de muchísimos otros proyectos y referencia para los programadores de C de todo el mundo: el Gnu GCC (http://gcc.gnu.org) . Para instalarlo bastará con que lo busques en tu gestor de paquetes preferido. No obstante, esta herramienta funciona por línea de comandos -”la ventanita negra”-, con lo que su uso y configuración puede llegar a ser farragoso e incluso complicado. Para facilitar la vida a los programadores de C, existen varios IDEs para programar en C de forma cómoda. Uno de ellos es CodeBlocks (http://www.codeblocks.org) , el cual se puede descargar con Gnu GCC incorporado o sin él (si ya estaba instalado previamente), descargándose entonces sólo lo que sería el IDE propiamente dicho. Otro IDE libre, algo más pesado, sería Eclipse (http://www.eclipse.org). En Windows disponemos de un “port” (es decir, una versión) no oficial de GCC llamado MinGW (http://www.mingw.org), que al igual que éste, funciona por línea de comandos. Si queremos IDEs de C para Windows, podemos volver a recurrir a CodeBlocks, ya que es multiplataforma, y podremos volver a elegir cuando lo descarguemos si deseamos sólo el IDE o bien el IDE más el compilador incorporado (que en Windows será MinGW). También podríamos elegir Eclipse, ya que también es multiplataforma, o también el IDE libre Dev-Cpp (http://www.bloodshed.net) .Hay que decir también que Microsoft ofrece un entorno de programación en C/C++ llamado "Visual C++Express Edition",que aunque no es libre, se puede descargar gratuitamente desde http://msdn.microsoft.com/vstudio/express/downloads/

Las instrucciones de compilación vienen especificadas en el archivo README presente en el interior del paquete tar.gz. Tal como se explica allí, para poder compilar Bennu se necesitan tener previamente instaladas las dependencias del entorno que ya conocemos (SDL, SDL_Mixer -y las que ésta arrastra: Smpeg, Ogg, etc...-, LibPng y Zlib), lo cual, en Linux es tan fácil como tirar del gestor de paquetes pero que en Windows hay que ir a las diferentes webs de las diferentes librerías y descargarse una a una su versión compilada para Win32. Además, evidentemente, para poder compilar Bennu se deberá disponer de un compilador de C, elemento que ya hemos comentado que en Linux será el Gnu GCC y en sistemas Windows el MinGW (más el entorno de consola MSYS para poder introducir los comandos de compilación, ya que el intérprete de comandos estándar de Windows -el “cmd.exe”- no sirve). Una vez cumplidos los requisitos previos, la compilación de Bennu se realiza de la forma clásica:

Para generar el archivo Makefile que guardará toda la información sobre la arquitectura de la máquina, dependencias resueltas, etc y que servirá en el paso siguiente para realizar la compilación de la forma más optimizada posible para nuestra máquina en concreto.

```
./configure
```

En el paso anterior se pueden introducir múltiples parámetros para personalizar hasta el extremo la posterior compilación (por ejemplo, con el parámetro --prefix=/ruta/instalación (muy típico) se puede especificar la ruta de instalación que deseamos, diferente de la por defecto, etc, etc). Para compilar propiamente, con la información obtenida en el paso anterior:

```
make
```

Una vez terminada la compilación, podríamos instalar Bennu (como usuario "root") con este comando:

```
sudo make install
```

Por otra parte, existe otra forma diferente de "bucear" dentro del código fuente de Bennu: consultar el repositorio SVN de Bennu en Sourceforge. Me explico. Sourceforge es un portal que ofrece espacio de almacenamiento, alojamiento web y muchos más servicios (estadísticas, foros, historiales, etc) a multitud de proyectos libres, facilitando así a los programadores de todo el mundo la puesta en marcha de un sitio en Internet exclusivo para su software. El código de Bennu se aloja en Sourceforge, y por ello, dispone de los servicios que este portal ofrece. Y uno de ellos es precisamente el repositorio SVN, el cual se encarga de almacenar todo el código fuente, pero no sólo eso, sino que guarda todas las versiones anteriores de todos los ficheros que se han ido modificando a lo largo del tiempo, sabiendo en todo momento quién y cuándo se modificó determinada línea de determinado fichero. Con este sistema se obtiene un gran control de la evolución de un código fuente, pudiendo dar marcha atrás en cualquier momento. En la página oficial de Bennu, en el apartado de descargas, el enlace "Repo browser" nos lleva al repositorio SVN, donde podremos observar los últimos cambios acumulados (que tal vez todavía no hayan sido trasladados a los instaladores oficiales) correspondientes a los ficheros de código fuente que queramos. De todas maneras, aunque mediante el navegador acabamos de ver que podemos acceder a un repositorio SVN, lo más recomendable para acceder a él es utilizar algún cliente SVN específicamente.

GNU/Linux ofrece multitud de clientes SVN para utilizar, pero Windows no tiene ninguno por defecto, así que tendremos que recurrir algún cliente gratuito. Uno libre (y multiplataforma, así que también lo podemos usar en el sistema del pingüino es el TortoiseSVN (http://subversion.tigris.org ). Una vez instalado, lo único que tendrás que hacer para acceder al código fuente de Bennu es ir a al menú "Admin"->"Commandline" y escribir el siguiente comando:

```
svn co https://bennugd.svn.sourceforge.net/svnroot/bennugd bennugd
```

## *Funcionamiento básico de Bennu

Una vez descargado el entorno Bennu, y según ya hemos visto, en GNU/Linux dispondremos del compilador */usr/bin/bgdc* y del intérprete */usr/bin/bgdi*, y en Windows dispondremos de los ejecutables correspondientes *bgdc.exe* y *bgdi.exe* copiados dentro de la carpeta elegida por nosotros en el proceso de instalación. Ambos archivos son programas que se ejecutan en un intérprete de comandos: ejecutándolos con un doble clic de ratón no hará nada, ya que necesitan unos parámetros determinados para que hagan su función, como luego veremos.

Vamos a explicar un poco la razón de ser de estos dos archivos. Tal como se explica en el Apéndice A, el código Bennu primero se compila y posteriormente, a la hora de ejecutarlo, se interpreta. Es decir, el proceso a la hora de escribir y ejecutar un programa en Bennu es el siguiente:

1º) Escribimos el código Bennu de nuestro juego con cualquier editor de texto plano (el "Bloc de Notas" en Windows o el Gedit de Gnome, el Leafpad de LXDE o similar en GNU/Linux), y guardamos ese código fuente en un archivo que ha de tener la extensión **.prg**

2º) Seguidamente, abrimos un intérprete de comandos *(en Windows, hay que ir a Inicio->Ejecutar y escribir "cmd.exe" -sin comillas-; en GNU/Linux existen infinidad de maneras, consulta la ayuda de tu distribución)* y con el comando "cd" del sistema (para Windows y para Linux) nos situamos en la carpeta

donde hemos grabado el archivo prg anterior. Una vez allí, compilamos este archivo .prg. Esto se hace (tanto en Windows como en GNU/Linux) escribiendo en el intérprete de comandos la siguiente orden:

```
bgdc nombrearchivo.prg
```

Obtendremos como resultado un nuevo archivo, (en la misma carpeta donde estaba nuestro código fuente),de formato binario, cuyo nombre es idéntico al del fichero .prg y cuya extensión se genera automáticamente y es **.dcb**

3º)Una vez obtenido este archivo **.dcb**, cada vez que queramos ejecutar nuestro juego tendremos que interpretar dicho archivo. Esto se hace teniendo abierta la línea de comandos y situándonos otra vez en la misma carpeta donde se ha generado el archivo .dcb. Desde allí podremos escribir la siguiente orden:

```
bgdi nombrearchivo.dcb
```

¡Y ya está!, esto es todo lo que necesitas para programar en Bennu

Nota: no es imprescindible moverse a la misma carpeta donde están los archivos .prg y .dcb. También se pueden compilar/interpretar si se introduce la ruta completa (absoluta ó relativa) de dichos archivos. En Windows sería por ejemplo algo así como *bgdc x:\ruta\donde\esta\mi\prg\archivo.prg*, donde x es la letra que sea, y en GNU/Linux algo muy similar, como *bgdc /ruta/donde/esta/mi/prg/archivo.prg*.

Resumiendo: primero compilamos con *bgdc* los archivos con extensión .prg (donde está escrito el código fuente) y luego, si queremos poner en marcha el juego, ejecutamos con *bgdi* el archivo con extensión .dcb obtenido de la compilación previa.Por lo tanto, podemos concluir que el compilador de Bennu deja el programa en un especie de pseudo-código (¡que no es código maquina!) y luego el intérprete de Bennu (diferente para cada plataforma) ejecuta el programa. De hecho, por eso Bennu es multiplataforma, porque existe un intérprete diferente para tres sistemas diferentes: Windows, Linux y Wiz. Sin embargo, uno se podría preguntar que, si lo que se pretende es ser multiplataforma, por qué es necesario el compilador previo, ya que se podría interpretar directamente los códigos fuentes y ya está. La respuesta es que hay varias razones para seguir la estructura existente: la primera es que al haber compilado el código previamente, la entrada que recibe el intérprete está mucho más optimizada y le facilita mucho su trabajo, con lo que esto redunda en la velocidad e integridad de la ejecución de los juegos. Y otra razón es que gracias al compilador, si no te interesa enseñar el código fuente por ahí –aunque eso no es libre-, puedes repartir tu juego simplemente ofreciendo el archivo .dcb, sabiendo que seguirá siendo igual de portable.

## *Mi primer programa en Bennu

Bueno, se acabó la teoría. Vamos a crear ya nuestro primer programa en Bennu, realizando uno a uno los pasos detallados en el apartado anterior.

1º) Abre tu editor de texto plano preferido y escribe el siguiente código (con las tabulaciones tal como están):

```
Import "mod_text";

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end

end
```

Un detalle importante antes de continuar es saber que Bennu es lo que se llama un lenguaje "case-insensitive"; es decir, que es igual que escribas todas las palabras del código en mayúsculas o minúsculas, o alternándolas si quieres: es lo mismo escribir "while" que "While" que "whILe" que "wHiLe"... Esto, que parece una tontería, en algunos lenguajes como C o Java no es así y hay que ir con cuidado con ellos. Pero Bennu no tiene ese "problema".

2º) Guarda el texto que has escrito en un archivo con extensión .prg, en la carpeta que quieras. Ahora abre la línea de comandos de Windows. Acuérdate del comando "cd" para moverte a la carpeta donde tienes tu código guardado. Una vez allí, pues, escribe **bgdc <miprograma>.prg**, donde <miprograma> es el nombre de fichero para tu .prg.Si todo va bien verás un listado del resultado de la compilación , o un mensaje de error si ha habido algún problema.

*¿Qué significa este mensaje obtenido por pantalla después de realizar la compilación? ¿Cómo sé si se ha producido algún error o no? Bien, si se produce algún error, observarás cómo Bennu te lo indica con un mensaje que muestra el código prg que contiene el error, la línea concreta del código donde está situado y una frase indicando el tipo de error que se ha producido. Será nuestro trabajo saber interpretar esa frase para entender qué tipo de error existe e intentar subsanarlo: normalmente los mensajes son bastante entendibles si se tiene una mínima experiencia con Bennu. Por ejemplo, un mensaje de error que nos podríamos encontrar podría ser éste:* ***C:\ruta\arxivo.prg:9: error:Procedure name expected ("END")*** *, donde se nos dice que en la línea 9 de arxivo.prg se ha producido un error determinado (en concreto,se nos ha olvidado abrir el END.)O este otro:* ***C:\ruta\arxivo.prg:27:error:Undefined procedure ("WRITE")****, donde el error es diferente (se nos ha olvidado escribir una orden "import") y se sitúa en la línea 27.*

*Si la compilación se produce con éxito, lo veremos inmediatamente porque aparece en pantalla un listado aportando diferentes datos que el compilador ha recopilado a lo largo de su ejecución. Estos datos ahora mismo no nos dicen nada, pero más adelante pueden sernos de utilidad: podemos observar el número de procesos y de variables globales, locales, privadas y públicas que tenemos definidas en nuestro código (en el capítulo 4 estudiaremos todos estos elementos), la cantidad de memoria que ocupan éstas y el código entero, el número de cadenas, estructuras e identificadores usados, etc. En definida, información de compilación que solamente con mostrarse podremos concluir que se ha realizado todo el proceso correctamente.*

*Si eres observador, habrás visto que aunque la compilación se realice con éxito, antes del listado (casi) siempre aparece un mismo mensaje, parecido a los mensajes de error:* ***C:\ruta\librender:0:warning:Variable redeclared ("ALPHA_STEPS")****.Como podrás intuir, el archivo que contiene el código que provoca este "error" no es el tuyo, sino uno llamado librender, que forma parte del entorno Bennu. De hecho, no es un error, es un "warning" (una advertencia, una notificación), que no impide en absoluto que la compilación se realice correctamente: simplemente está informando que librender tiene definida una variable -la llamada ALPHA_STEPS- más de una vez. Esto a nosotros como programadores Bennu no nos importa ni nos afecta lo más mínimo, así que haremos siempre caso omiso de este mensaje, como si no existiera, y podremos concluir que la compilación efectivamente se ha realizado con éxito.*

3º)Ya has compilado, pero... ¿dónde está tu juego? Ya sabemos que el compilador genera un fichero resultado con la extensión .dcb y con el mismo nombre que el fichero .prg que se acaba de compilar,en la misma carpeta donde estaba el fichero prg. Sin movernos, pues, de esa carpeta, ahora ejecutaremos nuestro programa usando el intérprete de Bennu desde la línea de comandos así: **bgdi <miprograma>** (no es necesario especificar la extensión .dcb cuando se invoca al intérprete)

Si todo va bien tu programa se estará ejecutando Has hecho que aparezca la frase "Hola mundo" moviéndose con scroll de un lado a otro. Y todo esto con tan sólo este código tan corto. Si no te funcionó, asegúrate de haberlo escrito todo correctamente y haber elegido las rutas correctamente.

Existe otra manera alternativa de poder compilar y ejecutar tus programas sin tener que recurrir a la consola de comandos, y sin tener que grabar los *.prg en una carpeta determinada. Basta con arrastrar con el ratón el icono que representa el archivo prg que has escrito, desde donde lo hayas guardado, y soltarlo sobre el icono que representa el bgdc (tendrás que tener una ventana abierta con el contenido de la carpeta "bin" de Bennu visible). De esta manera, obtendrás igualmente el archivo dcb. Y para ejecutar, tendrás que arrastrar el archivo dcb y soltarlo sobre el icono del bgdi. No obstante, a lo mejor, después de haber arrastrado y soltado el archivo prg sobre bgdc intentas buscar el archivo dcb y no lo encuentras. Esto indica que algo has escrito mal en el código, porque el compilador no ha

sido capaz de generar el dcb. El pequeño hándicap de este método es que si quieres saber cuál ha sido tu error, no puedes, ya que el mensaje de error (indicando el tipo y en qué linea del código se ha producido) aparece impreso dentro del intérprete de comandos, con lo que sólo trabajando desde allí lo podremos ver en pantalla y rectificar.

Respecto a las tabulaciones del texto, has de saber que no son en absoluto necesarias para que la compilación se produzca con éxito. Simplemente son una manera de escribir el código de forma ordenada, clara y cómoda para el programador, facilitándole mucho la faena de "leer" código ya escrito y mantener una cierta estructura a la hora de escribirlo. Respetar las tabulaciones forma parte de la "Guía de Estilo de Codificación Bennu" (de la cual se hará mención más detenidamente en el capítulo siguiente), y que es un documento que promueve una convención estándar de escritura de código Bennu para que todos los programadores tengan un estilo similar y para que este código sea mucho más fácil de leer entre varias personas.

Y tranquilo: pronto explicaré qué es lo que hace cada una de las líneas del programa, paso a paso: este ejemplo simplemente era para que vieras cómo funciona la dinámica de editar, compilar e interpretar.

## *Programas de soporte al desarrollador de Bennu

La distribución oficial de Bennu sólo incluye los ficheros y utilidades comentadas en los apartados anteriores. Nada más. No hay un IDE oficial, ni viene incluido ningún tipo de editor gráfico o de sonido, algún generador de animaciones o explosiones ni ninguna herramienta similar, típicas de otros entornos de desarrollo de videojuegos. La alternativa ante este hecho es o bien programarse uno todas las herramientas que necesite (con la consiguiente pérdida de tiempo y energía) o bien recurrir a otros programas de terceros (ver Apéndice B para más información). Afortunadamente, dentro de la comunidad Bennu han aparecido diversos proyectos que pretenden cubrir diferentes aspectos no resueltos por el entorno Bennu oficial, facilitando así la labor del programador, haciéndola mucho más cómoda y amigable. A continuación presento una lista de las utilidades que muy probablemente te serán imprescindibles cuando quieras realizar un videojuego de mediana entidad.

***IDE FLAMEBIRD MX*** (https://sourceforge.net/projects/fbtwo)

Ya es visto el proceso que se ha de seguir para escribir, compilar e interpretar nuestro programa. Y estarás de acuerdo en que es bastante incómodo utilizar estas herramientas independientes por línea de comandos. Por eso, algunas personas han creado por su cuenta varios IDEs (recuerda que un IDE es básicamente un editor de código con un montón de funciones útiles -corregir códigos, compilar y ejecutar, y muchas funciones más-), pero sin duda el IDE para Bennu más completo que puedes encontrar (aunque sólo para Windows) es el FlameBird MX.

Una vez descargado el FlameBird y descomprimido su contenido en una carpeta, no hay que hacer nada más. No hay que instalar nada. Dentro de la carpeta ya viene el ejecutable listo para abrir el IDE. Y ya está. Si quieres, puedes crearte un acceso directo desde el menú Inicio. Una vez iniciado el programa, antes de nada ves al menú "File->Preferences" y allí selecciona la sección "Global->File". Selecciona de la lista qué tipo de archivos se deberían asociar con FlameBird para que al hacer doble click en alguno de ellos se abriera el IDE automáticamente. Selecciona ".PRG", de momento, y ".FBP" también. Y también tienes que seleccionar donde dice "Open .dcb files with Bennu interpreter": esto es para poder interpretar el juego –y por lo tanto, ejecutarlo- desde dentro del propio entorno FlameBird, sin tener que salir fuera a ejecutar el intérprete.

Otra cosa que hay que hacer una vez iniciado por primera vez el programa es decirle justamente en qué carpeta tienes instalado el entorno Bennu, para que el FlameBird sepa dónde ir a buscar el compilador y el intérprete cada vez que lo necesite, sin tener que hacerlo tú. Para eso tienes que ir otra vez al menú "File->Preferencesr", y allí seleccionar la opción "Compiler": podrás especificar la ruta de la carpeta de Bennu. Y ya tendrás el FlameBird listo para usar.

Verás que arriba tienes los iconos típicos (Nuevo -código fuente, proyecto-, Abrir, Guardar,Copiar, Pegar, Compilar y ejecutar, etc.) En medio tienes el espacio para programar. A la izquierda controlas todos los ficheros que necesitará tu programa. A la derecha verás todas las variables,procesos e identificadores que usa tu programa para que puedas tenerlo todo a mano. Abajo aparecen los textos de error al compilar, te acostumbrarás a ello con el uso.Puedes probar de escribir, grabar el prg en una carpeta cualquiera y ejecutar el mismo programa de antes. Verás que es mucho más cómodo porque lo tienes todo a mano.

Mediante FlameBird podrás crear archivos individuales de código fuente, o bien proyectos, que vienen a ser conjuntos

de varios archivos de código fuente relacionados entre sí (cuando un programa empieza a ser complejo es conveniente dividir su código fuente en diferentes archivos). Si creas un nuevo proyecto, verás que aparece un nuevo tipo de fichero aparte de los PRG y DCB: los ficheros **FBP.** Estos archivos son los archivos de proyectos del FlameBird (si eliges este IDE), y son los que guardan toda la información sobre las relaciones entre los distintos códigos fuente de un mismo proyecto y la configuración general del proyecto. Cuando hagas un proyecto, el archivo que tendrás que seleccionar para abrir todo el proyecto, con sus diferentes PRGs, en el FlameBird será éste (acuérdate de que hace cuatro párrafos hemos asociado esa extensión al FlameBird). En un proyecto se seguirá generando un único DCB.

***RESALTADORES DE SINTAXIS PARA VARIOS EDITORES***
(http://forum.bennugd.org/index.php?topic=165.0)
(http://forum.bennugd.org/index.php?topic=165.15)

A falta de un IDE oficial y multiplataforma diseñado específicamente para Bennu, podemos recurrir a IDEs genéricos, agnósticos de ningún lenguaje en concreto, y adaptarlo a nuestras necesidades. En concreto, en los enlaces anteriores podemos descargar una serie de parches para hacer compatibles con Bennu unos cuantos IDEs libres de programación, para que puedas elegir el que más te guste y adaptarlo.

La forma de instalación es siguiendo estos pasos:
1) Instalar el IDE oficial.
2) Desempaquetar en la carpeta de instalación del IDE los archivos con el parche. En algunos se requerirá ejecutar algunas acciones adicionales. Se detallarán según el caso.
3) Colocar los ejecutables Bennu y sus dlls en la carpeta "C:\Bennu\Bin\".

Todos tienen resaltado de sintaxis, compilación y ejecución integrada en el editor, visualización de salida de compilación. Algunos adicionalmente tienen templates básicos de creación del programa (estilo "helloworld") y/o atajos de inserción de trozos de código pre-tipeados.

**Crimson Editor** (http://www.crimsoneditor.com )
- Excelente editor para Windows, con un manejo de columnas idéntico a Ultraedit.
- Sintaxis de colores
- Template "hello world"
- Ctrl+F1 = Compile,  Ctrl+F2 = Run

**ConTEXT** (http://www.contexteditor.org )
- Para Windows
- Sintaxis de colores
- Atajos de inserción de trozos de código pre-tipeados
- F9 = Compile, F10 = Run
- Se debe ejecutar el archivo "fixbennu.reg" (disponible en los enlaces de descarga) para completar la instalación

**PSPad editor** (http://www.pspad.com/es)
- Para Windows
- Sintaxis de colores
- Template "hello world"
- Atajos de inserción de trozos de código pre-tipeados
- Ctrl+F9 = Compile,  F9 = Run
- Se debe ejecutar el archivo "PSPad-Bennu-Install.bat" (disponible en los enlaces) para completar la instalación

NOTA: En este editor no es posible capturar la salida de ejecución, pero sí la de compilación.

**Ultraedit** (http://www.ultraedit.com )
- Para Windows
- Sintaxis de colores
- Botones de compilación y ejecución
- Listado de funciones y procesos contenidos en el prg

NOTA: Leer el archivo "install.txt" contenido en el paquete.

**Gedit** (http://projects.gnome.org/gedit)
- Editor de texto plano oficial del escritorio Gnome
- El fichero “bennugd.lang” es el fichero de especificación para el resaltado de sintaxis con colores. Se ha de copiar en la carpeta  /usr/share/gtksourceview-2.0/language-specs, arrancar el programa y en el menú “View->Highlighting” elegir “BennuGD”.
- También se puede descargar el fichero “bennugd-mime.xml” que sirve para que GNU/Linux reconozca los archivos de extensión prg como del tipo MIME text/x-bennugd (un tipo inventado arbitrariamente): básicamente este tipo MIME es una derivación del text/plain, pero debido a que el archivo “bennugd.lang” anterior hace referencia a este tipo MIME, es preferible descargarlo, y copiarlo en la carpeta /usr/share/mime/packages para posteriormente ejecutar

el comando *update-mime-database /usr/share/mime* ,para que GNU/Linux se entere de la incorporación al sistema de un nuevo tipo de fichero.
-Gedit tiene una opción del menú que sirve para ejecutar comandos externos. Se puede utilizar para invocar desde allí el compilador y/o intérprete sin tener que salir al terminal.

Otros editores libres (y multiplataforma) genéricos interesantes, que podrían ser objeto de algún parche futuro para dar soporte al lenguaje Bennu son los siguientes:
**-Kwrite** (http://kate-editor.org/kwrite) -editor de texto plano oficial del escritorio KDE-
**-Geany** (http://www.geany.org)
**-Scite** (http://www.scintilla.org/SciTE.html)
**-Notepad2** (http://www.flos-freeware.ch/notepad2.html)
**-Notepad++** (http://www.flos-freeware.ch/notepad2.htm). Es el editor elegido para el BennuPack
**-Jedit** (http://www.jedit.org)
**-Xint** (http://www.xtort.net/xtort-software/xint)
**-Editra** (http://editra.org)

***FPGEDIT 2009*** (http://forum.bennugd.org/index.php?topic=619.0)

Ahora no es el momento de explicar cómo funciona un editor FPG ni sus características, lo haremos en el capítulo 3. Baste con decir que un FPG es un fichero-paquete que incluye dentro de él los gráficos que se deseen para nuestro programa. El "FPG Edit" es un de gestor de FPGs: permite crearlos e incluir en ellos las imágenes que usaremos en nuestro juego, pudiéndolas clasificar, ordenar, editar sus puntos de control, grabar los FPGs en tres profundidades (8,16,32 bits), exportar una imagen concreta fuera del FPG, y muchas cosas más.

Si no éste, casi de forma imprescindible necesitarás utilizar un editor de FPGs. Ya hablaremos de ello cuando toque, pero es importante señalar que éste (o alguno similar) serán programas que más adelante usaremos mucho, así que si quieres, lo puedes instalar ya para ver cómo es.

***FNTMAKE 2009*** (http://forum.bennugd.org/index.php?topic=697.0)

Es un creador de fuentes de letra para nuestro juego, en un formato específico de Bennu (FNT). Puede trabajar con fuentes de 1,4 y 32 bits. Ya hablaremos de él cuando toque, pero lo puedes instalar ya, porque lo usaremos mucho en este curso.

***EDITOR FPG de PRG*** (http://forum.bennugd.org/index.php?topic=267.0)

Otro editor de FPGs, que tiene la particularidad de haber sido programado íntegramente en Bennu,y por tanto, ser multiplataforma. Con este programa se pueden crear, cargar y grabar FPGs de 8,16 y 32 bits, añadir y eliminar gráficos, editar puntos de control y otras propiedades de los gráficos del interior del FPG,

Además, incluye un pequeño pero funcional editor de fuentes FNT.

***SMART FPG EDITOR*** (http://code.google.com/p/smartfpgeditor)

"Smart Fpg Editor" es otro potente, fácil y muy amigable editor FPG, pero sólo funciona en Windows. Con este programa se pueden crear, cargar y grabar FPGs de 8,16 y 32 bits, añadir y eliminar gráficos, editar los puntos de control y otras propiedades de los gráficos del interior del FPG.

***BENNUPACK*** (http://bennupack.blogspot.com)

Es posible que estés pensando que es un poco pesado tener que ir descargando tantos programas por separado desde tantos sitios diferentes: que si Bennu de un lugar, que si el Flamebird de otro, el FPGEdit y el FNTMake, etc. Esto mismo es lo que pensaron los chicos del grupo de desarrolladores Coldev, y lo que han hecho ha sido reunir en un único paquete instalable (sólo para Windows) todas estas herramientas -y más-, para disfrute y comodidad de los usuarios de este sistema operativo. Es lo que ellos llaman el BennuPack.

Es decir, en vez de tener que instalar una a una las diferentes herramientas, aspecto un poco pesado sobre todo para el principiante -y peor si no se conocen-, con el BennuPack dispondremos de un único fichero "Setup", el cual instalará (creando sus correspondientes accesos directos del menú Inicio de Windows y también un desinstalador accesible desde el menú "Agregar/quitar programas " del panel de control de Windows) los siguientes programas, todos ellos de una sola vez, como si fueran uno solo:

*Compilador e intérprete Bennu
*IDE Notepad++ con los parches para Bennu ya aplicados
*FPGEdit

*FNTMake
*Generador de sprites 2D Charas.EX (http://www.charas-project.net)
*Generador de explosiones Explogen (http://www.kenginegaming.com)
*Generador de paquetes autoinstalables PakAtor (http://semitex.iespana.es/)
*Generador y editor de tiles TileStudio (http://www.tilestudio.org)
*Editores gráficos Idraw3, EA Graphics Editor (http://www.nba-live.com/eagraph), SplitImage (http://www.thecastle.com) y Slowview (http://www.slowview.at)
*Multitud de códigos fuente de ejemplo, clasificados en tres niveles de dificultad (muchos de los cuales se han utilizado en este manual como soporte para las explicaciones). El nivel avanzado de ejemplos corresponde de hecho a juegos totalmente acabados y funcionales de todo tipo (lucha, carreras, plataformas,rpgs...).
*Multitud de librerías dll externas.Podemos encontrar la librería Bennu3D (http://3dm8ee.blogspot.com), desarrollada por el propio equip ColDev, y varias librerías más que aportan soporte 3D a Bennu, librerías de red como la Fsock o la Net, librerías de acceso a bases de datos como la Sqlite o la OpenDbx, librerías de soporte a la reproducción de vídeo, al reconocimiento de mandos y periféricos, librerías de efectos visuales, etc, etc, etc.

Si te has fijado bien, habrás podido comprobar cómo, de toda la lista anterior de programas, sólo uno, el editor Prg de FPGs es multiplataforma. Los demás, aunque son libres, sólo funcionan en Windows. Esto es debido a qué el lenguaje con el que están programados, o bien el uso que hacen de determinadas librerías, limitan su funcionamiento a dicho sistema operativo. No obstante, los usuarios de GNU/Linux pueden utilizar estos programas pensados para Windows mediante un emulador. Un emulador es un programa que permite que otros programas se ejecuten en una máquina como si en realidad se tratara de otra. En nuestro caso lo usaremos para ejecutar programas sobre Windows, cuando en realidad, la máquina real con Windows no existe.

El emulador libre más completo de Windows para GNU/Linux es Wine (http://www.winehq.org). Es muy fácil de instalar y utilizar. Para instalarlo, haremos uso de los repositorios oficiales de nuestra distribución favorita. En el caso de Debian por ejemplo, haríamos:

```
sudo aptitude install wine
```

A partir de aquí, lo único que debemos hacer es descargarnos los instaladores para Windows de los programas que queremos emular. Si estamos trabajando en el escritorio Gnome o bien KDE, probablemente haciendo doble click sobre el instalador (al igual que se haría en Windows), comenzará el familiar proceso de instalación. Así de fácil. Una vez acabado, encontraremos la entrada de nuestro programa bajo el menú “Aplicaciones->Wine”. Si el doble click no funcionara, siempre podremos abrir el intérprete de comandos y ejecutar lo siguiente para arrancar el instalador:

```
wine archivoInstalador.exe
```

y una vez instalado el programa, también podríamos ponerlo en marcha en el intérprete de comandos así:

```
wine "C:\Program Files\NombreDelPrograma\ejecutableDelPrograma.exe"
```

Finalmente, si quieres configurar el comportamiento de Wine, lo puedes hacer mediante el comando:

```
winecfg
```

Evidentemente, este programa tiene muchísimas posibilidades que no trataremos aquí. Investígalas.

## *Recursos web sobre Bennu

A continuación tienes una lista que reúne algunas de las más importantes direcciones de recursos web sobre Bennu existentes actualmente , para que la tengas como referencia a lo largo de todo el curso:

| | |
|---|---|
| http://forum.bennugd.org | Foro oficial de Bennu (en español e inglés) donde la comunidad participa activamente. Muy recomendable. La cantidad de información que se reúne en sus posts es de un valor incalculable. La asiduidad de sus miembros y la gran voluntad en aportar nuevas ideas y soluciones y ayudar a los más novatos lo |

| | |
|---|---|
| | convierten un lugar permanente de intercambio. A día de hoy, es la mejor fuente de información sobre Bennu. |
| http://wiki.bennugd.org | Referencia online del lenguaje. Poco a poco se está construyendo; con el tiempo vendrán explicadas todas las especificaciones sobre cada una de las funciones del lenguaje Bennu (qué parámetros usan, qué devuelven, para qué sirven, etc), sobre las diferentes variables (de qué tipo son, para qué sirven, qué valores pueden tener), sobre las palabras claves del lenguaje (construcción de bucles, condicionales, declaración de variables, creación de procesos,etc), incluso códigos de ejemplo que muestran el uso de las funciones.<br><br>Es completamente imprescindible que consultes y explores esta documentación oficial.<br>Aunque todavía no esté completa, pretende llegar a ser el centro de información central y actualizada del lenguaje. Ten en cuenta que el presente manual sólo abarca los conceptos más básicos de la programación, y no repasa ni mucho menos –tampoco es su objetivo- toda la funcionalidad Bennu que puede aportar. Para ampliar y profundizar en el conocimiento del lenguaje y de su entorno, la documentación es, pues, visita obligada.<br><br>De hecho, recomiendo descargar su contenido en el disco duro local para poderla tener más a mano. Para realizar la descarga de múltiples páginas web de un mismo sitio en GNU/Linux podemos utilizar la utilidad de consola Gnu Wget (instalada por defecto en la mayoría de distribuciones y si no,siempre presente en los repositorios oficiales; es muy recomendable consultar su manual para ver sus opcionesmás importantes) y en Windows podemos recurrir a la aplicación libre HTTrack (http://www.httrack.com).De entre las soluciones privativas (sólo para Windows) podemos recurrir a WebCopier (http://www.maximuxsoft.com) o incluso Acrobat (http://www.adobe.com ), el cual convertirá toda la referencia en un único pdf muy práctico. |
| http://www.bennugd.org | Portal oficial de Bennu, desde donde podrás acceder a las últimas descargas oficiales del entorno en sus múltiples plataformas y rastrear su código fuente, además de poder consultar las últimas noticias, consultar la wiki y entrar en el forum oficial. También dispone de una lista centralizada de links a diferentes proyectos no oficiales relacionados con Bennu, , algunos de los cuales han sido comentados en apartados anteriores (por ejemplo, el IDE FlameBird2, la librería Bennu3D ó la Fsock entre otras) |
| http://www.bennugd.co.uk | Foro de Bennu especialmente pensado para los angloparlantes |
| http://code.google.com/p/bennugd-wii | Port no oficial de Bennu para la consola Nintendo Wii |
| http://code.google.com/p/apagame4be | Proyecto que reúne diversos juegos clásicos, listos para ser ejecutados pero también incluyendo su código fuente, para que pueda ser estudiado. El ánimo del Apagame4be es pedagógico, incidiendo más en la sencillez y claridad de los códigos que no en su funcionalidad o extensión. Los juegos que están incluidos a fecha de hoy son: Tetris, Columns, DrMario, Pang, Muro, Pong, TicTacToe, Galaxian y Conecta4 |
| http://code.google.com/p/pixjuegos | Conjunto de juegos escritos en Bennu, con sus respectivos códigos fuente.<br>Para conocer más proyectos escritos en Bennu, se puede consultar el apartado de “Proyectos” del foro de Bennu. |
| http://div.france.online.fr/ | Portal de la comunidad francesa de Fénix/Bennu. Con bastante contenido en tutoriales y juegos. Existe un foro. |
| http://www.trinit.es/tutoriales | Espléndido tutorial de introducción al lenguaje Bennu. De nivel básico, pero abarca aspectos no tratados en este manual, como el uso de la librería Bennu3D. |

## *La distribución de tus programas

Todavía no hemos aprendido a programar ni un solo código nosotros solos, pero hay un detalle que es importante que sepas ya cómo va: el tema de pasar tus juegos a otras personas, y que les funcione en su ordenador. Cuando ya hayas acabado de desarrollar tu juego, y en tu ordenador vaya perfectamente, ¿qué es lo que tienes que dar a otra persona para que lo pueda disfrutar en su ordenador, el cual no tendrá ningún entorno Bennu previamente instalado?

Salta a la vista que lo primero imprescindible para que a otra persona le vaya el juego es darle el fichero .dcb y ADEMÁS, el intérprete adecuado a su plataforma para que lo pueda ejecutar: *bgdi.exe* para Windows y *bgdi* para GNU/Linux. Para ello, lo más habitual es incluir estos ficheros necesarios (de momento ya tenemos dos, pero ahora veremos que habrá más), dentro de un paquete tipo Zip, Tar.gz o similar. De esta manera, el usuario del juego recibirá un sólo archivo dentro del cual, una vez descomprimido, se encontrarán todos los archivos necesarios para poder ejecutar el juego sin problemas.

Existen bastantes programas compresores-empaquetadores libres y gratuitos que permiten trabajar con multitud de formatos de compresión, entre los cuales está el formato casi omnipresente Zip, pero también el formato RAR, el ACE, etc además de sus formatos propios. En tu distribución de GNU/Linux es seguro que tendrás más de un programa de este estilo ya instalado de serie sin que tengas que hacer nada, consulta la ayuda pertinente. En Windows, programas de este tipo que puedes utilizar pueden ser:7-Zip (http://www.7-zip.org) o IZARC (http://www.izarc.org ).

Ya hemos comentado anteriormente que el intérprete de Bennu necesita a su vez de otros archivos para poder funcionar: son las dependencias externas (SDL, SDL_Mixer, etc). Por lo tanto, además de incluir en el paquete tipo Zip el fichero .dcb y el bgdi adecuado a la plataforma en la que queramos ejecutar el juego, deberás incluir también todas estas librerías:

*Si tu juego está pensado para funcionar en Windows, hacer esto es muy fácil: simplemente tienes que copiar (atención, exactamente en la misma carpeta donde esté ubicado el intérprete del juego) todas las librerías ("todas" es una generalización: si tu juego ni reproduce audio Vorbis no necesitarás las librerías correspondientes, etc, pero obviaremos estos detalles) que hay bajo la carpeta "externals" presente dentro de la ruta de instalación de Bennu. Y ya está.

*Si tu juego está pensado para funcionar en GNU/Linux, este aspecto es un poco más farragoso, debido a que, a diferencia de Windows, el entorno Bennu no viene con ninguna carpeta "externals" para copiar. Si recordamos cómo se instalaron estas dependencias externas, tuvimos que recurrir a nuestro gestor de paquetes preferido para instalarlas de forma independiente. Así que cuando pasemos nuestro juego a otra persona, ésta deberá hacer lo mismo previamente: deberá instalarse "a mano" estas dependencias previas (si es que no lo están ya, cosa por otro lado bastante probable en las distribuciones de hoy en día). O bien intentar crear nosotros algún tipo de "shell script" que automatice la tarea de comprobar si las dependencias se cumplen, y si no, descargarlas e instalarlas...el problema de esto último es que cada distribución de Linux funciona de forma diferente respecto la manera de descargar e instalar paquetes, así que deberíamos tener en cuenta muchas casuísticas diversas en este nuestro hipotético "shell script".

*Este hecho de no incorporar de serie en nuestro juego para GNU/Linux las dependencias necesarias puede ser un poco incómodo, para el jugador,realmente. Por eso puedes probar un método para incluir en nuestro paquete las dependencias externas "a la manera de Windows" (que no siempre funciona, todo hay que decirlo, debido a las diferencias de arquitectura de paquetes entre diferentes distribuciones). Se trata de recopilar uno a uno los ficheros físicos que se corresponden con cada una de las dependencias externas (que nosotros como desarrolladores del juego sí tenemos al tener instalado el entorno Bennu), y copiarlos a nuestro paquete Zip. Básicamente, se trata de buscar qué paquetes concretos tenemos instalados de las diferentes librerías (SDL, SDL_Mixer, LibPng, Zlib, etc) y seguidamente, buscar qué fichero concreto dentro de esos paquetes es el que nos interesa copiar a nuestro paquete Zip.La manera concreta de hacerlo varía según la distribución, pero por ejemplo, en distribuciones basadas en el formato de paquetes .deb (como Ubuntu ó Debian) se haría de la siguiente manera:*

*Para recordar el nombre exacto de los paquetes externos de los que depende Bennu (por ejemplo, el de SDL):*

```
aptitude search sdl
```

*Aparecerá una lista de paquetes que incluyen en su nombre "sdl". En nuestro ejemplo, tenemos una distribución Ubuntu y hemos encontrado el paquete "libsdl1.2debian-all". En esta lista también se puede ver si ese paquete está o no instalado en nuestro sistema en este momento: si la primera letra de la línea es "p", no está instalado; si es "i", sí lo está. En el caso de que no lo estuviera, procederemos a instalarlo.*

```
sudo aptitude install libsdl1.2debian-all
```

*Finalmente, lo que nos interesa: buscar la librería concreta que se nos acaba de instalar con el paquete (ya que el paquete incorpora muchos tipos de ficheros: documentación, recursos, etc, que no nos interesan para nada ahora).En nuestro ejemplo, esto se hace con el comando (¡"-L" ha de ser mayúscula!):*

```
dpkg -L libsdl1.2debian-all
```

*El fichero que nos interesa normalmente estará ubicado en la carpeta /usr/lib y tendrá extensión .so (o bien .so seguida de varios números separados por punto). Lo único que deberemos hacer es copiar todos los que veamos que tengan dicha extensión dentro de nuestro paquete del juego.*

*Otra forma de conseguir estos ficheros .so es, en vez de obtenerlos a partir de la instalación del paquete correspondiente, compilarlos uno mismo a partir del código fuente de cada una de estas librerías externas en cuestión. Pero este tema ya requiere unos conocimientos elevados de programación en C que se escapan de los objetivos de este texto.*

Todavía nos quedan más cosas por incluir en el paquete de nuestro juego. En párrafos anteriores hemos comentado que Bennu está modularizado, esto es: que por una serie de ventajas que ya hemos explicado, la funcionalidad de Bennu está separada y organizada en diferentes ficheros-módulos, ubicados dentro de la carpeta "modules" (tanto en la instalación de Windows como la de GNU/Linux). Resulta bastante obvio pues, que también deberemos incluir dentro de nuestro paquete los módulos que hayamos hecho servir en nuestro código, copiando los módulos necesarios de la carpeta "modules" del entorno Bennu. (evidentemente, los módulos .dll de Windows para que nuestro juego funcione en Windows y los módulos .so de GNU/Linux para que nuestro juego funcione en ese sistema) Pero no sólo eso, sino que también deberemos copiar las librerías internas presentes en la carpeta "libs" del entorno Bennu (como con los módulos, las librerías a copiar serán las .dll de Windows o las .so de GNU/Linux según nos interese) que sean necesarias para que los módulos funcionen correctamente, puesto que(como ya hemos comentado) existen algunos módulos (no todos) que dependen de determinadas librerías internas de Bennu, por lo que evidentemente deberemos de incluir las que sean necesarias. ¿Y cómo sabremos de qué librerías depende un determinado módulo que hayamos utilizado en nuestro código? Pues mediante la utilidad *moddesc*, que describiremos en el siguiente capítulo.

*Respecto el tema de la inclusión de librerías (tanto las externas como las internas y módulos) en juegos para GNU/Linux hay que tener un detalle extra claro. En los juegos para Windows no hay ningún problema: copiamos todos los ficheros dll necesarios dentro del paquete y el juego ya se podrá ejecutar inmediatamente, pero en GNU/Linux, el sistema ha de saber dónde se encuentran las librerías *.so que nuestro juego usará: no basta con copiarlas en la misma carpeta que nuestro juego, hay que hacer un paso extra. Este paso extra es el que, de hecho, realiza el instalador del entorno Bennu cuando copia sus propias librerías en el disco duro,genera el archivo /etc/ld.so.conf.d/bennugd.conf y ejecuta el comando ldconfig. No obstante, como este mecanismo es relativamente complejo, nosotros a la hora de distribuir nuestro juego para GNU/Linux haremos otra cosa para indicar al sistema dónde se encuentran las librerías requeridas por nuestro juego: modificar el valor de la variable de entorno LD_LIBRARY_PATH. Si no sabes lo que es una variable de entorno del sistema, no te preocupes: para lo que sirve LD_LIBRARY_PATH es precisamente para indicarle a Linux en qué rutas ha de ir a buscar las librerías *.so que cualquier programa requiera al ejecutarse. Y esto se hace abriendo el intérprete de comandos, situándonos en la carpeta donde se encuentra nuestro juego con todas sus librerías -comando "cd"- y ejecutando seguidamente la siguiente orden:*

```
LD_LIBRARY_PATH=.;$LD_LIBRARY_PATH
```

*Esta línea en concreto lo que hace es establecer el valor de LD_LIBRARY_PATH , que será la ruta de la carpeta actual (el "." significa eso: carpeta actual, sea cual sea, por eso es importante habernos situados previamente en la carpeta que toca, la que contiene todas las librerías) además del valor que pudiera tener anteriormente la propia variable.*

*La diferencia entre utilizar la variable LD_LIBRARY_PATH y utilizar el método que usa el instalador de Bennu en GNU/Linux es que este segundo método es permanente y el primero no. Es decir, con ldconfig el sistema sabrá siempre a partir de entonces dónde ir a buscar las librerías, aunque reinicie, pero con LD_LIBRARY_PATH, cada vez que, no ya que se reinicie el sistema, sino que se cierre y se vuelva a abrir un intérprete de comandos, deberemos volver a actualizar su valor al adecuado, porque se habrá perdido. Para solucionar este engorro, se puede crear con cualquier editor de texto plano un pequeño bash shell script (llamado "runme.sh", por ejemplo) que incluya primero la línea anterior, y seguidamente lanzara el intérprete de Bennu para ejecutar el juego, y utilizar siempre este shell script para poner en marcha nuestro programa. Por ejemplo,*

```
#!/usr/bash
LD_LIBRARY_PATH=.;$LD_LIBRARY_PATH
bgdi juego.dcb
```

Finalmente, si tu juego tiene imágenes –las tendrá- ,es evidente que las tendrás que meter también dentro del paquete que quieres distribuir; así que tendrás que incluir todos los archivos FPG que hayas utilizado,o bien, si en tu código has cargado las imágenes individualmente –con *load_png* por ejemplo, tendrás que incluir en el paquete una a una las imágenes PNG por separado (del tema imágenes ya hablaremos más adelante). Además, si incluyes sonidos, también tendrás que meterlos dentro del paquete a distribuir, incluyendo pues cada uno de los archivos WAV,OGG,MOD,PCM,XM… individuales. Lo mismo si has utilizado videos, o cualquier otro tipo de archivo auxiliar.

Es importante que te acuerdes de poner (dentro del paquete a distribuir) todos los videos,sonidos e imágenes en las direcciones relativas que usaste en la carga de éstos dentro del código fuente. Es decir, si pusiste *load_fpg("gráficos.fpg")* tendrás que guardar el archivo graficos.fpg en la misma carpeta que el DCB, pero si pusiste *load_fpg("imagenes/graficos.fpg")* debes crear una carpeta dentro de la que tienes el DCB llamada "imágenes" y meter en ella el archivo "graficos.fpg". Esto lo entenderás mejor cuando veamos el tema de la carga de imágenes –o video o sonido, es igual-.

Otro detalle muy importante es que en otros sistemas operativos no Windows no es lo mismo "Uno" que "uno", es decir, que distinguen entre mayúsculas y minúsculas, y es necesario que al escribir las direcciones donde tienes los ficheros te asegures que escribes las mayúsculas y minúsculas de la misma manera.

Ya tienes el paquete creado con todos los archivos necesarios en su interior. Lo importante del tema ahora es que si entregas por ejemplo tu programa en un Zip que incluye tu DCB, más los archivos de imágenes/sonidos/videos, más el intérprete más las librerías necesarias, el usuario final que lo único que quiere es jugar, no tendrá ni idea de lo que tiene que hacer cuando reciba este conjunto de archivos.¿Dónde clica?¿Dónde está el ejecutable del juego? El único ejecutable que aparece es el intérprete, pero clicando en él sólo saldrá un pantallazo negro…

Una solución es programar un shell script en GNU/Linux o un archivo BAT en Windows que realice automáticamente todos los pasos necesarios para poner en marcha nuestro programa, y que sea el único fichero que el usuario tenga que invocar. No obstante, sale fuera de los objetivos de este texto exponer las técnicas de programación de shell scripts para Bash (Linux) o de archivos por lotes para Windows: recomiendo leer algún manual específico, o buscar información en la documentación de tu sistema o en Internet

Otra solución sería la siguiente: renombrar el intérprete *bgdi* poniéndole el mismo nombre que el del fichero .dcb ( pero manteniendo su extensión EXE en el caso de Windows, y sin extensión en el caso de GNU/Linux). Es decir, si tienes inicialmente el archivo DCB que se llama "Pepito.dcb", hay que hacer que el intérprete de Bennu se llame ahora "Pepito.exe" en Windows ó "Pepito" a secas en GNU/Linux. Y ya está. El jugador potencial hará clic en el único ejecutable que ve y ahora sí, el juego se ejecutará sin más problema. Incluso podrías cambiarle además la extensión al archivo DCB y ponerle la aséptica extensión DAT , y funcionará también. Puedes probarlo, si quieres, con nuestro primer programa Bennu que hemos visto y un amigo con su ordenador.

De todas maneras, si eres un usuario de Windows, supongo que estarás pensando que distribuir tu juego dentro de un simple archivo Zip no es la manera más habitual en este sistema operativo. A tí te gustaría que tu juego se pueda distribuir en un único ejecutable "Setup", y que la gente, al clicar en él, instalara el juego en su ordenador como cualquier otro programa: con un asistente que copiara automáticamente todos los archivos a una ruta especificada por el usuario, que fuera preguntando si se desea una entrada en el menú Inicio de Windows o si quiere crear un acceso directo

en el escritorio,etc...el procedimiento habitual en una instalación de cualquier programa en Windows.Para conseguir esto, necesitas utilizar algún programa diseñado para esto: los generadores de instalaciones. Aquí tienes una lista –incompleta y limitada- de algunos de estos programas para Windows que te pueden ayudar, por si te los quieres mirar.

| |
|---|
| **Nsis** (http://nsis.sourceforge.net) Es libre |
| **Wix** (http://sourceforge.net/projects/wix) Es libre |
| **Inno Setup Compiler** (http://www.jrsoftware.org) Es freeware (gratis pero no libre) |
| **Setup Generator** (http://www.gentee.com/setupgen) Es freeware |
| **AdminStudio** (http://www.macrovision.com ) -dentro del apartado FLEXnet InstallShield- |
| **Advanced Installer** (http://www.advancedinstaller.com) |
| **CreateInstall** (http://www.createinstall.com) |
| **InstallConstruct** (http://www.mg-india.com) |
| **Installer VISE** (http://www.mindvision.com) |

Finalmente, comentar un último detalle: ya has visto que el icono del intérprete de Bennu es precisamente un especie de ave fénix con corona, y si renombramos el archivo como nuestro nuevo juego, nuestro juego tendrá como icono ese mismo ave. Existe una manera de cambiar este icono y poner el que nosotros queramos, pero de eso hablaremos en otro capítulo posterior, cuando comentemos precisamente el uso de este tipo de imágenes.

## *Stubs y otras opciones del compilador de Bennu

Un aspecto curioso respecto el tema de la distribución de nuestro juego es que disponemos de la posibilidad de crear "stubs" (también conocidos como "mochilas"). Y esto ¿qué es? Pues generar a partir de nuestro archivo .prg directamente un archivo .exe. Este archivo exe incorporará en su interior el intérprete *bgdi* más el código dcb del juego. Es decir, que en vez de tener el intérprete por un lado y el código compilado por el otro, podremos tener las dos cosas juntas (el intérprete "con mochila") dentro de un ejecutable. Esto se hace en el proceso de compilación (de igual manera en Windows y en GNU/Linux: usaremos nuestro compilador *bgdc* de forma normal, pero ahora haciendo uso de un parámetro concreto , el -s, así:

```
bgdc -s bgdi nombreDelJuego.prg
```

Lo que hemos hecho aquí es indicar que al archivo generado en la compilación (lo que hasta ahora había sido un archivo DCB) se le va a "incluir dentro" un ejecutable, indicado justo después del parámetro -s, que en este caso (y siempre) será el intérprete *bgdi*. El resultado será un ejecutable llamado nombreDelJuego.EXE.

Pero todavía podemos ir más allá: podemos incluir dentro de este archivo nombreDelJuego.EXE a más a más todos los recursos que utilicemos en el juego (archivos de sonido, imágenes, videos, archivos FPG, etc, etc), de tal manera que podamos distribuir nuestro juego en forma de un único archivo ejecutable (más, eso sí, todas las librerías y módulos necesarios aparte, ya que éstos no se pueden incluir en ningún ejecutable de ninguna manera) .Ésta sería una manera posible de dificultar la apropiación de parte de otras personas de los archivos de recursos del juego -músicas, imágenes,etc-. Para lograr esto, tenemos que escribir:

```
bgdc -a -s bgdi nombreDelJuego.prg
```

Fíjate que simplemente hemos añadido el parámetro -a. Un detalle importante es que, para que esta opción funcione, todos los recursos que se incrusten en el ejecutable han de estar dentro de carpetas cuyas rutas relativas especificadas dentro del código fuente tienen que ser las mismas que respecto el directorio donde está ubicado el compilador (es decir, se toma la carpeta donde está *bgdc* como directorio base). Cuidado con esto.

Insisto que con la opción -a (y la -s también, obviamente) hay un tipo de archivos -sólo ése- que no se incluye dentro del ejecutable: las librerías (DLL en Windows, SO en GNU/Linux). Esto es así por la propia naturaleza de estas librerías. Así que, si utilizas la opción -a (junto con la -s) lo que obtendrás finalmente será un ejecutable único,

pero necesitarás incorporar además en el paquete tipo Zip las librerías necesarias para que el juego se pueda ejecutar correctamente.

Finalmente, existe otra opción interesante del compilador, sustitutiva de -a, que es la opción -f seguida del nombre/ruta de un fichero, así:

```
bgdc -f graficos.fpg -s bgdi nombreDelJuego.prg
```

Esta opción sirve para lo mismo que la opción -a (es decir, para incluir ficheros dentro del ejecutable resultante de la compilación) pero sólo para incluir un fichero concreto ( el especificado despúes de -f). Es decir, en vez de incluir todos los ficheros de recursos, sólo se incluye uno, el que se le diga. También se pueden añadir varios archivos nombrándolos tras -f uno a uno separados por un espacio en blanco.

Por otro lado,el compilador de Bennu tiene más opciones interesantes que no tienen que ver con stubs: puedes ver el listado completo si escribes simplemente *bgdc* y nada más en la línea de comandos. Para empezar, podemos descubrir en el listado de opciones que nos saca esta ayuda, la opción -g, la cual funciona así:

```
bgdc -g nombreDelJuego.prg
```

y que sirve para guardar información de depuración en el DCB para permitir de esta manera ejecutar una consola especial mientras esté en marcha nuestro juego (sin el parámetro -g no lo podríamos hacer) que nos deje visualizar el estado de la ejecución, de los procesos, mostrar y/o cambiar valores de las variables, estructuras, etc. Hablaremos mucho más detenidamente de la consola de depuración en capítulos posteriores.

También disponemos de la opción -i, la cual funciona así:

```
bgdc -i rutaCarpeta nombreDelJuego.prg
```

y que sirve para añadir la ruta de la carpeta especificada al PATH del sistema, por si fuera necesario.

Y la opción -o , la cual sirve simplemente para especificar detrás de ella el nombre del fichero .dcb que queramos que tenga, en vez del nombre por defecto (que es el que ya tiene el fichero .prg). Es decir, si queremos que a partir de nombreDelJuego.prg se genere automáticamente el fichero nuevoNombreDelJuego.dcb (en vez de nomobreDelJuego.prg), tendríamos que hacer lo siguiente:

```
bgdc -o nuevoNombreDelJuego.dcb nombreDelJuego.prg
```

Finalmente, comentaremos tres opciones avanzadas que en estos momentos no sabremos cómo ni por qué utilizar, pero que detalladas aquí nos servirán para que, según avancemos en este curso, podamos recurrir a ellas cuando las necesitemos. Cuando hayas terminado de leer este texto y vuelvas otra vez aquí, entonces lo entenderás perfectamente. Una de estas opciones bastante compleja es la opción -d:

```
bgdc -d nombreDelJuego.prg
```

que sirve para mostrar por pantalla gran cantidad de información completa sobre el resultado de la compilación, útil sobretodo para la depuración del propio entorno Bennu, así que normalmente el programador estándar no hará uso de esta opción. El mismo parámetro también existe en el intérprete, *bgdi*, para obtener información completa sobre el resultado de la interpretación de nuestro programa, en tiempo real (también de un nivel muy avanzado).

Otra opción diferente a la anterior es la opción -D (¡no confundir con -d!). Esta opción ha de venir acompañada detrás de ella por una pareja *macro=valor* la cual se le enviará al compilador. Una macro es una constante que se puede definir desde el intérprete de comandos, y sirve para muchas cosas. Por ejemplo, paraa habilitar o deshabilitar partes de tu código, o seleccionar funcionalidades diferentes en tu proyecto (por ejemplo, generar un proyecto en modo “demo” o modo “release”). También para poder definir constantes que servirán luego en tu código; por ejemplo, se puede definir ANCHO_PANTALLA=800 para luego poder usar en tu código ANCHO_PANTALLA en vez de poner 800. Las macros son como las líneas #define que veremos en el capítulo 4, pero sin la posibilidad de usar parámetros. Para definir una macro en tiempo de compilación se haría así:

```
bgdc -D PEPITO nombreDelJuego.prg
```

y luego en el código escribir:

```
#ifdef PEPITO
...
#endif
```

Con esto se podría usar código que sólo se ejecutara en modo “debug” sólo cuando se compila en modo “debug”. Otro ejemplo sería compilar diferentes versiones sin tener que tocar el código (sería como un #define pero sin estar escrito en código). Otro posible ejemplo sería:

```
bgdc -D PEPITO=100 nombreDelJuego.prg
```

para poderlo usar en código así

```
array[PEPITO]=”Última posición en el array”
```

Finalmente, también podríamos enviar al compilador la opción avanzada -Ca, la cual permite las declaraciones de procesos automáticamente. Muy útil cuando no queramos declararlos nosotros en el código (con la sentencia “Declare”, que estudiaremos también en el capítulo 4), pero tiene un descenso de efectividad en tiempo de ejecución. Existe una opción (-p) que es justamente la contraria: obliga a declarar absolutamente todos los procesos del código.

```
bgdc -Ca nombreDelJuego.prg
```

## *Mi segundo programa en Bennu

¿Te has quedado con las ganas de escribir alguna otro programa y ver qué hace? Venga, prueba con éste (respeta las tabulaciones):

```
Import “mod_text”;
Import “mod_rand”;

Process Main()
private
          int mivar1;
end
begin
    loop
          delete_text(0);
          mivar1=rand(1,10);
          if (mivar1<=5)
               write(0,200,100,2,"Menor o igual que 5");
          else
               write(0,200,100,2,"Mayor que 5");
          end
          frame;
     end
end
```

Prueba de ver si eres capaz de intuir qué es lo que hace este código, aún sin saber nada del lenguaje. Comprueba lo que ocurre al ejecutarlo. Por cierto,si tienes problemas al cerrar el programa, no te preocupes, insiste: tal como está escrito este código es normal.

# CAPITULO 2

## Empezando a programar con Bennu

### Explicación paso a paso de "Mi primer programa en Bennu"

Recordemos el código que utilizamos para ver ese scroll tan bonito de la frase “Hola mundo”:

```
Import “mod_text”;

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end
end
```

Vamos a ir viendo poco a poco el significado de las diferentes líneas de este código, el cual nos servirá para introducir diferentes conceptos importantes sobre programación en general y programación en Bennu en particular.

### *Primera línea: sobre la importación de módulos. Utilidad "moddesc"

La primera línea que encontramos en el código es *"Import 'mod_text'";*. Esta línea lo que hace (es fácil intuirlo) es importar en nuestro programa el módulo mod_text. Pero, ¿esto qué quiere decir?. Tal como se explicó en el capítulo anterior, un módulo es una librería separada del resto que contiene cierta funcionalidad específica muy concreta. Podemos encontrar el módulo Bennu para el manejo del teclado, el módulo Bennu que se encarga del dibujado de sprites, el módulo Bennu encargado de la gestión del sonido, etc. Dicho de otra forma, los comandos del lenguaje Bennu están organizados y separados por módulos, de manera que un comando concreto está “ubicado” dentro de un módulo concreto. Esto implica que para hacer uso de un comando particular, deberemos incluir el módulo que lo contiene, porque de lo contrario Bennu no sabrá qué significa ese comando. Fíjate por cierto que a la hora de incluir el módulo, no se especifica ningún tipo de extensión: en Windows los ficheros que forman los módulos tienen extensión ".dll" y en GNU/Linux su extensión es ".so", pero precisamente para hacer que nuestro código se pueda compilar e interpretar en cualquier plataforma, las extensiones no se toman en cuenta.

Fíjate también que esta línea (como algunas otras más) acaba en punto y coma. Esto será importante recordarlo más adelante.

Así pues, como en el código de ejemplo hemos utilizado dos comandos (que son **delete_text()** y **write()** , enseguida los estudiaremos) que están definidos dentro del módulo "mod_text", debemos incluir éste dentro de nuestro programa. De esta manera, al hacer referencia a dicho módulo, podremos utilizar todos los comandos que hay en él definidos: de lo contrario, el entorno Bennu no sabría qué hacer cuando leyera **delete_text()** o **write()** porque no sabría lo que son. Esto último es fácil de comprobar: borra la línea del "Import" de nuestro ejemplo e intenta volver a compilar el programa. ¿Qué es lo que ocurre? Pues que en el intérprete de comandos aparece un error diciendo algo así como *"Undefined Procedure ("DELETE_TEXT")"*, indicándonos además el nombre del fichero fuente donde ha aparecido el

error y en qué número de línea. Si te fijas, el mensaje de error no es ni más ni menos que lo que acabamos de comentar: al llegar el compilador a la línea donde aparece un comando que está definido en un módulo pero al no estar éste importado, el compilador se encuentra con código que no entiende y no sabe qué hacer, abortando la compilación.

En este ejemplo, sólo ha sido necesario importar un módulo, pero es mucho más habitual tener que importar entre cinco y diez módulos, que aportan cada uno un conjunto de funcionalidades concretas. Por cada módulo que necesitemos importar deberemos de escribir una línea "Import" diferente. Todas las líneas Import que se escriban deberán de aparecer una detrás de otra al principio del código fuente que las requiera, antes de cualquier otro elemento.

Existe otra manera diferente de importar módulos que no es mediante la palabra "Import". Se trata de crear, en la misma carpeta donde esté ubicado nuestro fichero fuente .prg, un fichero de texto plano llamado obligatoriamente igual que dicho fichero fuente .prg pero que tenga la extensión ".imp" ó bien también vale ".import". En este archivo .imp incluiremos los nombres de los módulos que queramos importar (sin incluir ninguna extensión ni comillas), uno por línea. Es decir, este nuevo archivo no es más que una lista de módulos a cargar en nuestro proyecto.Por ejemplo, en el ejemplo anterior, si a nuestro archivo .prg lo hemos llamado "Ejemplo1.prg", deberíamos crear en su misma carpeta un archivo de texto llamado "Ejemplo1.imp" que contendrá una única línea que será `mod_text`.

También existen los archivos ".imp" globales, los cuales serán siempre tenidos en cuenta indistintamente para todos los archivos .prg que se generen. De esta manera se puede configurar una lista de módulo precargados por defecto para cualquier proyecto que se realice. Estos archivos ".imp" globales deberán tener obligatoriamente el nombre de "bgdc.imp" y/o "bgdc.import" y deberán estar ubicados en el directorio del compilador.

La idea con estos archivos ".imp" no es sólo evitar tener que poner constantemente múltiples líneas "Import" en diferentes códigos fuente, sino que también sirve para modificar rápidamente en un momento determinado la lista de módulos importados sin tener que editar los ficheros fuente.

Tener modularizado el entorno Bennu es una buena idea por una serie de razones que detallamos en el capítulo anterior, pero añade un pequeño engorro en el que seguramente ahora estarás pensando y preguntándote: ¿cómo se puede saber a qué módulo pertenece un determinado comando que deseo utilizar? Bien, la respuesta a esta pregunta la da la utilidad de línea de comandos *moddesc* (en Windows, ubicada en la carpeta "bin" junto con el compilador e intérprete, y en GNU/Linux ubicada en /usr/lib/bgd). Esta aplicación recibe como parámetro el nombre de un módulo (sin extensión otra vez) y nos saca por pantalla un montón de información útil relativa a ese módulo en concreto, entre la cual está qué comandos incluye dentro de él. Más concretamente, la forma de utilizarlo sería así (por ejemplo, con el módulo "mod_text":

```
moddesc mod_text
```

Si ejecutas la línea anterior, verás que aparece por pantalla (tras el texto del copyright) una serie de datos separados por secciones, las cuales vamos a repasar. La primera sección ("Module name:") simplemente indica cual es el nombre del archivo físico que se corresponde con el módulo en cuestión; en Windows ya sabemos que será un fichero con extensión .dll y en GNU/Linux un fichero con extensión .so. La siguiente sección ("Constants:") indica cuáles son las constantes que podremos utilizar en nuestro código si importamos dicho módulo -dicho de otra manera: cualquiera de las constantes que aparecen listadas no las podremos usar en nuestro código si no importamos el módulo en cuestión-. El concepto de constante y su uso lo veremos en el capítulo 4. Seguidamente viene la sección "Functions:"), que precisamente es la que nos indica qué comandos vienen incluidos en el módulo especificado, y por tanto, en el caso de estudiar el módulo "mod_text", allí veremos los comandos **delete_text()** y **write()**, entre otros. Otra sección interesante es la que se titula "Module dependency:", la cual nos muestra de qué librerías depende el módulo estudiado, si es que depende de alguna (puede ser que no dependa, esto es, cuando todo su código binario esté contenido en el propio módulo sin tener que recurrir a otros ficheros); en el listado sólo aparecen las librerías internas propias de Bennu -las que están en la carpeta "lib"-, no las dependencias externas. Por otro lado, en el caso del módulo "mod_text" no ocurre, pero si escogemos algún otro módulo,pueden aparecer más secciones, como por ejemplo la sección "Globals:", la cual indica qué variables globales se podrán utilizar en nuestro código siempre que importemos el módulo adecuado (el concepto de variable global y su uso también lo veremos en el capítulo 4).

De todas maneras, es posibles que pienses que es un poco pesado tener que ejecutar todo el rato este comando para saber en un momento concreto qué comando pertenece a qué módulo. ¿No habría la posibilidad de tener en un único fichero impreso toda la información que extrae *moddesc* acerca de todos los módulos, para así poderla consultar de una forma mucho más rápida y cómoda? Pues sí, pero este fichero con toda la información de todos los módulos lo tendremos que generar nosotros, “a mano”. Por suerte, con una simple orden del intérprete de comandos

(tanto en Windows como en GNU/Linux) podremos realizar lo que nos proponemos, de la siguiente manera:

Para obtener en Windows un fichero llamado “moddesc.txt” (aunque, evidentemente, se puede llamar como queramos) deberemos primero situarnos -mediante el comando “cd”, recordemos- en la carpeta donde están ubicados los módulos (es decir, la carpeta “modules” bajo la carpeta de instalación de BennuGD), y desde allí ejecutar:

```
for %i in (mod_*.dll) do moddesc %i >> moddesc.txt
```

En GNU/Linux, para obtener el mismo fichero “moddesc.txt”, deberemos hacer lo mismo: situarnos en la carpeta donde están los módulos de Bennu (en este caso, la carpeta “/usr/lib/module”) y ejecutar lo siguiente:

```
for i in mod_*.so ; do moddesc $i >> moddesc.txt ; done
```

En ambos casos (tras pulsar varias veces “Enter”), obtendremos el fichero “moddesc.txt” que nos servirá de mucha ayuda a la hora de consultar qué comando, constante o variable están definidas en qué módulo.

## *La línea "Process Main()"

En esta línea es cuando realmente empieza el programa. TODOS los programas que escribamos deberán de comenzar (después de haber realizado los “Import” pertinentes, o bien después de algún posible comentario) con esta línea siempre. Esta línea señala el comienzo de lo que técnicamente se llama “proceso principal”. En el capítulo 4 se profundizará ampliamente en el concepto de proceso, pero por ahora lo único que has de saber es que esta línea es la puerta de entrada que el intérprete de Bennu necesita para poder empezar a ejecutar el código.

Ya veremos posteriormente que de procesos pueden haber muchísimos en un mismo programa, y todos comienzan con la palabra reservada “process”, pero lo que distingue el proceso principal de los demás es precisamente su nombre: “Main()”, el cual es obligatorio que sea exactamente así (los otros procesos que puedan existir pueden tener el nombre que nosotros deseemos). Este nombre específico es el que hace que se distinga este proceso principal -el único obligatorio- de los posibles otros procesos que puedan existir.

Hemos dicho que “process” es una palabra “reservada”. Por “palabra reservada” se entiende cualquier palabra o comando propio e intrínseco del lenguaje Bennu,(es decir, es un nombre que está reservado como palabra clave del lenguaje de programación) y que por tanto, se ha de escribir tal cual –aunque recordad que Bennu es “case-insensitive”-, y no se puede utilizar para ninguna otra función que no sea la que Bennu tiene prevista para ella. Precisamente por ser comandos intrínsecos de Bennu, las palabras reservadas no necesitan ser importadas por ningún módulo, ya vienen de “serie”, por decirlo así. Es el caso de “Process”, así como también de todas las sentencias declarativas, condicionales o iterativas que veremos a lo largo de este capítulo.

## *Comentarios

Un comentario es simplemente una nota aclaratoria sobre el programa. Los comentarios no son necesarios para el correcto funcionamiento del programa, aunque son muy útiles para entender su funcionamiento. Sirven sobre todo para explicar dentro del código fuente partes de éste que puedan ser más o menos complejas. Hay dos tipos de comentarios:

*De una sola línea: comienzan con el símbolo // y terminan al final de la línea en que se definen.
*De varias líneas: comienzan con el símbolo /* y terminan con el símbolo */. También pueden también comenzar y terminar en la misma línea.

Todos los textos incluidos en un comentario son ignorados por el compilador. Se pueden poner tantos comentarios en un programa como sea necesario, y en cualquier punto del programa.

Por ejemplo, nuestro programa se podría haber escrito:

```
Import "mod_text";

//Esto es un comentario de una línea.
Process Main()

/*Y éste
comentario es
un comentario
de varias*/
Private
          int mivar1;
End
Begin
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end
end
```

## *Bloque "Private/End"

Los ordenadores tienen una memoria en la que pueden almacenar valores. Una "variable" es una posición concreta de la memoria del ordenador que contiene un valor determinado que es utilizado en un programa. Ese valor puede ser un número entero, decimal, un carácter, una cadena de caracteres,etc. En su programa el desarrollador puede asignar a las variables un nombre, como por ejemplo "x"; y entonces, en ese programa "x" se podrá utilizar para referirse al valor que contiene.

Declarar una variable es "crear" esa variable; es decir: reservar un espacio de la memoria del ordenador con un nombre, disponible para almacenar allí un valor determinado. Declarar una variable es como dejar sitio en el cajón de la habitación, y ponerle una etiqueta, para saber que ese lugar está reservado para colocar tus calcetines, por ejemplo.

En Bennu se delimita muy claramente la parte del programa que se encarga de declarar variables –al principio-, de la parte del código a ejecutar propiamente dicho. Primero de declaran las variables que pretendemos usar, y una vez hecho esto, se comienza el programa en sí. Evidentemente, si no se van a utilizar variables, no hace falta escribir su sección de declaraciones correspondiente

Es muy importante saber que hay cuatro clases de variables claramente diferentes en Bennu según sus prestaciones y objetivos -y excluyentes entre sí- : las variables privadas , las variables públicas, las variables locales y las variables globales. Cuando hablemos de procesos en el capítulo 4 se explicará la diferencia entre estas clases y el porqué de tal importante distinción, pero ya mismo podrás intuir que el bloque "Private/End" es la sección que se encarga de declarar en su interior variables de tipo privadas. A lo largo de todo este capítulo y el siguiente, será éste el único tipo de variables que utilizaremos, así que ahora no nos preocuparemos por los otros tipos.

## *Línea "int mivar1;". Tipos de datos en Bennu

Esta línea es la que propiamente declara una variable,llamada *mivar1*,y además dice de qué tipo es.

El nombre de la variable nos lo podemos inventar absolutamente (aunque hay algunas excepciones exóticas: no podemos darle un nombre que sea igual a algún comando existente de Bennu -por razones obvias: habría confusión-; el nombre no puede empezar por un número; ni tampoco puede incluir espacios en blanco –lógicamente, porque Bennu no sabría qué es la segunda palabra-, etc).

El tipo de la variable indica qué contenido es el que la variable está preparada para contener. Bennu necesita saber siempre qué tipo de valores puede o no albergar una variable determinada. No es lo mismo que una variable sea declarada para poder guardar números enteros que para guardar cadenas de caracteres. Si declaramos una variable como entera, si quisiéramos asignarle un valor de cadena de caracteres daría un error. Así pues, siempre, cuando se declara la variable, hay que decirle que será de un tipo determinado, y que por tanto, los valores que incluirá, sean los que sean, siempre serán de ese tipo determinado. Dependiendo de lo que queramos hacer con esa variable, nos convendrá crearla de un tipo o de otro, según lo que tengamos previsto almacenar en ella.

Los tipos de variables disponibles en Bennu son:

| | |
|---|---|
| **BYTE** | Almacenan valores numéricos enteros que pueden valer entre 0 y 255. Estos valores extremos también se pueden expresar con las constantes ***MIN_BYTE*** y ***MAX_BYTE***,respectivamente.<br>Una variable de este tipo necesita 8 bits de memoria (sin signo y sin coma decimal) para almacenar ese valor. |
| **SHORT** | Almacenan valores numéricos enteros que pueden valer entre -32767 y +32768. Estos valores extremos también se pueden expresar con las constantes ***MIN_SHORT*** y ***MAX_SHORT***,respectivamente.<br>Una variable de este tipo necesita 16 bits de memoria (con signo y sin coma decimal) para almacenar ese valor. |
| **WORD** | Almacenan valores numéricos enteros que pueden valer entre 0 y 65535. Estos valores extremos también se pueden expresar con las constantes ***MIN_WORD*** y ***MAX_WORD***,respectivamente.<br>Una variable de este tipo necesita 16 bits de memoria (sin signo y sin coma decimal) para almacenar ese valor. |
| **INT** | Almacenan valores numéricos enteros que pueden valer entre -2147483648 y +2147483647.Estos valores extremos también se pueden expresar con las constantes ***MIN_INT*** y ***MAX_INT***,respectivamente.<br>Una variable de este tipo necesita 32 bits de memoria (con signo y sin coma decimal) para almacenar ese valor. |
| **DWORD** | Almacenan valores numéricos enteros que pueden valer entre 0 y 4294967296. Estos valores extremos también se pueden expresar con las constantes ***MIN_DWORD*** y ***MAX_DWORD***,respectivamente.<br>Una variable de este tipo necesita 32 bits de memoria (sin signo y sin coma decimal) para almacenar ese valor. |
| **FLOAT** | Almacenan valores numéricos decimales, de poca precisión.<br>Estos números decimales son del tipo llamados "con coma flotante"; es decir: el número de decimales no es fijo porque puede variar en función de las operaciones matemáticas que se realicen con ellos. En Bennu, el tipo DOUBLE es sinónimo de FLOAT.<br>A la hora de escribir un número float , el símbolo utilizado para separar la parte entera de la parte decimal es el punto: "."<br>Una variable de este tipo necesita 32 bits de memoria (con signo y con coma decimal) para almacenar ese valor. |
| **STRING** | Almacenan cadenas de texto (una serie de varios caracteres que también puede estar vacía), de longitud variable -potencialmente ilimitada-, y que estarán comprendidas entre comillas dobles o simples.<br>Una variable de este tipo necesitará más o menos bits de memoria para almacenar un valor según la cantidad de caracteres que tenga esa cadena: más corta la cadena, menos espacio necesitará. |
| **CHAR** | Almacenan un un único carácter (una letra, signo, espacio o carácter especial de control).<br>Es posible realizar operaciones numéricas con un carácter o asignar su valor a una variable de tipo entero, con lo que se extrae el valor ASCII del carácter.<br>Una variable de este tipo necesita 8 bits de memoria para almacenar este valor; por lo tanto, hay 256 ($2^8$) posibles caracteres diferentes reconocidos por Bennu. En realidad, internamente este tipo de datos es indistinguible del tipo BYTE. |
| **POINTER** | Hacen referencia a una dirección de memoria donde está alojada una variable, NO al valor contenido en ella. Su uso requiere conocimientos avanzados de programación. |

También podríamos crear nosotros, como programadores que somos, nuestros propios tipos de datos, con el bloque “TYPE/END”, pero este tema lo abordaremos más adelante.

Llama la atención que haya tantos tipos de datos diferentes para almacenar números enteros. Esto es porque si sabemos que los valores de una variable nunca van a ser mayores que 255 por ejemplo, sería una tontería declarar una variable de tipo INT porque se estaría desperdiciando gran cantidad de memoria del ordenador para nada. Es como si tuviéramos un gran cajón del cual sólo utilizamos una pequeña parte. Debemos ajustar el tipo de las variables a los valores que prevemos que van a contener para ajustar recursos y no despilfarrar memoria con variables que la requieren pero que luego no se aprovecha.

Otra cosa importante: los tipos de dato enteros tienen un rango limitador: cuando sumas o restas una cantidad que provoca que el número salga fuera del rango que admiten, los bits más significativos del nuevo número se pierden. En otras palabras menos técnicas: se provoca que el número "dé la vuelta": si a una variable WORD que contiene 65535 le sumas 1, el valor resultante es 0 en lugar de 65536, ya que 65536 queda fuera del rango admitido por los tipos WORD.

Todos los datos de tipo entero vistos en el apartado anterior permiten especificar como prefijo **SIGNED** o **UNSIGNED** para obtener versiones de los mismos con o sin signo. Esto equivale a usar un tipo de dato entero del mismo número de bits pero con la salvedad del signo. Por ejemplo, UNSIGNED INT equivale exactamente a DWORD. La única variante de este tipo que no tiene una palabra clave como alias es **SIGNED BYTE**, un entero de 8 bits que puede contener un número entre -128 y 127. Estos valores extremos también se pueden expresar con las constantes ***MIN_SBYTE*** y ***MAX_SBYTE***,respectivamente.

Es posible que alguna vez os encontréis con algún código donde se declaren las variables sin especificar su tipo. Es una práctica que yo no recomiendo, pero basta saber que Bennu asume por defecto que esa variable es tipo INT.

Vemos claramente pues, volviendo a nuestro ejemplo, que la variable “mivar1” es de tipo INT. Para definir el tipo de una variable cuando se declara, pues, es siempre

```
tipo nombre;
```

Opcionalmente, a la vez que escribimos esta línea donde declaramos la variable y definimos su tipo, podríamos también rellenar de valor la variable, lo que se dice inicializar la variable –darle su valor inicial-. Si la declaramos y ya está, automáticamente su valor queda establecido por Bennu a 0 (cero). Para hacer esto, simplemente es de la forma

```
tipo nombre=valor;
```

En el ejemplo no lo hemos hecho así, pero perfectamente podríamos haber escrito

```
int mivar1=10;
```

y habernos ahorrado la línea posterior *mivar1=10;* (escribiendo pues en una sola línea la declaración y la inicialización de la variable), ya que de hecho lo único que hace esa línea precisamente es darle un valor a “mivar1” (ahora lo veremos).

Por otro lado, es posible que alguna vez os encontréis con algún código (en este texto, por ejemplo) donde se declaren varias variables del mismo tipo en una única línea. Es decir, en vez de escribir por ejemplo:

```
byte mivar1;
byte mivar2;
int mivar3;
```

podemos encontrarnos con

```
byte mivar1, mivar2;
int mivar3;
```

El significado de ambos códigos es exactamente el mismo: *mivar1* y *mivar2* es BYTE y *mivar3* es INT. La elección entre un tipo de escritura y el otro es puramente personal: declarar cada variable en una línea diferente ayuda a la claridad del código (de hecho, es la práctica recomendada), pero declarar varias variables (siempre del mismo tipo, insisto) en una sola línea al programador le resulta a veces más cómodo y rápido.

Un último detalle que conviene comentar es el hecho de que Bennu tiene predefinidas una serie de variables que sirven para tareas concretas –normalmente son variables locales-. Esto quiere decir que no podremos crear nosotros variables que tengan el mismo nombre que estas variables "de sistema" para evitar confusiones. Cada una de estas variables predefinidas realizan una tarea específica diferente de las demás y su utilidad es muy concreta, como iremos viendo poco a poco. Por ejemplo, Bennu incorpora en su implementación el uso de una variable denominada "x", que ya veremos para qué sirve. Esta variable "x" la podremos usar siempre que necesitemos recurrir a las ventajas concretas que ésta nos puede ofrecer, pero hemos de procurar no llamar a ninguna de nuestras variables personales "x", porque se podría confundir con la predefinida, aunque no la estemos usando. Hay que decir que para utilizar las variables predefinidas no hay que declararlas ni nada. Profundizaremos en esto más adelante.

Fijarse que ahora volvemos a acabar la línea con punto y coma.

## *Bloque "Begin/End"

Después de las declaraciones de variables, viene el código del programa/proceso propiamente dicho –es decir, donde se escriben los comandos que se van a ir interpretando- . Este código siempre comienza con la palabra reservada BEGIN ("empezar"), tras la cual puede aparecer cualquier número de sentencias(=órdenes=comandos,etc), y siempre finaliza con la palabra reservada END ("acabar").

Por cierto, en nuestro código hay dos "ends": el del begin es el último de todos.

Según lo dicho, el mínimo programa que por lo tanto podríamos escribir en Bennu (que no use variables) y que funcionaría sería algo así como

```
Process Main()
Begin
End
```

Por supuesto, si lo ejecutas, no pasará nada, porque realmente este código no hace absolutamente nada.

Fíjate que esta vez las líneas BEGIN y END no acaban en punto y coma.

## *Línea "mivar1=10;"

Un programa puede poner en cualquier momento de su ejecución un valor concreto en una variable.

Para poner el valor numérico 123 en la variable "x" por ejemplo, el programa utilizará una sentencia (orden) como la siguiente, (si la variable no la hemos declarado previamente como de tipo entero, puede que dé error):

```
x = 123;
```

Los programas pueden también consultar el valor de las variables, o utilizarlos en diferentes expresiones; por ejemplo, para poner en una variable entera llamada "y" el valor de "x" más 10, se utilizaría una sentencia como la siguiente:

```
y = x + 10;
```

Antes de que el ordenador ejecutara esta sentencia, "y" podría tener cualquier valor pero, después de ejecutarla, la variable contendrá exactamente el resultado de sumarle 10 al valor numérico que tenga la variable "x" en ese momento.

Queda claro después de todo lo dicho, que la tercera línea de nuestro programa lo que hace es asignarle a una variable "mivar1" creada por nosotros el valor de 10. Con ese valor almacenado en memoria con el nombre de "mivar1" y que le hemos dado el valor 10 se supone que en líneas siguientes haremos algo.

Fíjate que ahora volvemos a acabar la línea con punto y coma.

*NOTA: aprovecho para comentar aquí que en Bennu los operadores aritméticos son los mismos que en la mayoría de lenguajes de programación: + para la suma, - para la resta, * para la multiplicación , / para la división y % para el módulo (que equivale al resto de una división –es decir, 12%5 =2, 27%4 =3, 3%2 =1, etc-).*

Un detalle muy importante que hay que tener en lo que respecta a estas operaciones aritméticas básicas es que el tipo de datos del resultado (int, word, byte, float...) vendrá dado por el tipo de datos de los operandos. Es decir, que una suma de dos números Int dará un Int, la multiplicación de dos números Float dará un Float, etc. Esto es especialmente delicado cuando estamos realizando estas operaciones (división, multiplicación, etc) entre números decimales y sin darnos cuenta asignamos el resultado a una variable de tipo entera. Lo que ocurrirá es que el resultado se trunca sin redondear, por lo que si éste era por ejemplo 32.94, se convertirá en un 32; si era un 86.351 será un 86.

Cuando un operando es de un tipo (por ejemplo, byte) y el otro operando es de otro tipo (por ejemplo, int), el tipo del resultado será siempre igual a aquél tipo de los dos que esté por encima dentro de la jerarquía siguiente: float>dword>int>word>short>byte.

## *Bloque "While/End"

La sentencia WHILE ("mientras") es una sentencia que implementa un bucle, es decir, que es capaz de repetir un grupo de sentencias un número determinado de veces. Para implementar este bucle se debe especificar entre paréntesis la condición que se debe cumplir para que se ejecute el grupo de sentencias a continuación de la palabra reservada WHILE. Tras especificar esta condición, se pondrán todas las sentencias que se necesita que se repitan y, finalmente, se marcará el final del bucle con la palabra reservada END (no importa que dentro del bucle aparezcan más palabras END si éstas forman parte de sentencias interiores a dicho bucle).En nuestro ejemplo, el END correspondiente al While sería el segundo empezando por el final.

Así que la sintaxis genérica sería:

```
While (condición)
...
End
```

Cuando se ejecute la sentencia WHILE se realizará la comprobación que se especifica y, si ésta resulta cierta, se ejecutarán las sentencias interiores; en caso contrario, se continuara el programa a partir del END que marca el final del WHILE.

Si se han ejecutado las sentencias interiores, se volverá a comprobar la condición y, si ésta vuelve a ser cierta, se realizará otra iteración (se volverán a ejecutar las sentencias interiores). Se denomina iteración a cada ejecución del grupo de sentencias interiores de un bucle. Este proceso continuará hasta que, al comprobarse la condición del WHILE, ésta resulte falsa.

Si se llega por primera vez a una sentencia WHILE la condición resulta falsa directamente, entonces está claro que no se ejecutarán las sentencias interiores ninguna vez. Este detalle es importante tenerlo en cuenta.

Las sentencias interiores a un bucle WHILE pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles WHILE

En nuestro ejemplo concreto, tenemos el bucle:

```
while (mivar1<320)
       delete_text(0);
       mivar1=mivar1+2;
       write(0,mivar1,100,1,"¡Hola mundo!");
       frame;
end
```

Podemos comprobar que hay una condición –que "mivar1" sea menor que 320-, que es la que dirá si hay

que ejecutar o no (y si es que sí, hasta cuándo) las siguientes cuatro líneas. Justo en la línea de antes hemos visto que dábamos el valor de 10 a dicha variable. Pues bien, la condición del while no hace más que comprobar si el valor de esa variable "mivar1" es menor que 320. Como nada más llegar al while eso es verdad (10<320), se pasa a ejecutar una a una las sentencias incluidas dentro del bucle, hasta llegar al end. Cuando la ejecución del programa llega al end, se vuelve otra vez a la línea del while y se vuelve a comprobar la condición a ver si continúa siendo verdadera. Si "mivar1" continúa siendo menor que 320, se vuelve a ejecutar el interior del bucle, y así, y así hasta que llegue un momento que a la hora de efectuar la comparación otra vez, "mivar1" ya no valga menos que 320. En ese caso, el programa salta todas las líneas internas del bucle y continúa su ejecución justo después del end del while. Se supone que en algún momento durante la ejecución de las sentencias del interior del bucle, el valor de "mivar1" cambiará, porque si no cambiara, "mivar1" siempre valdría 10 y la condición siempre sería verdadera, por lo que siempre se ejecutarían las sentencias del interior del bucle y eso daría lugar a un bucle infinito: el programa no se acabaría nunca.

Fíjate que en nuestro programa cuando la condición sea falsa y el programa vaya a la siguiente línea después del end del while, se encontrará con que ya no hay nada más: (el end del programa); con lo que la ejecución del programa se acabará cuando la condición del while sea falsa.

Fíjate también que esta vez las líneas WHILE y END no acaban en punto y coma.

## *Orden "Delete_text()". Parámetros de un comando

La primera línea que nos encontramos dentro del bucle del while es el primer comando propiamente dicho que "hace algo". Este comando se encarga, como su nombre indica, de borrar de la pantalla todos los textos que haya en ese momento. ¡Pero si de momento no hay ningún texto que borrar: sólo le hemos dado un valor a una variable!, dirás. Es verdad, pero el texto lo escribiremos más adelante. Pero entonces, dirás otra vez: ¿por qué no escribimos el texto, y luego, si hay que borrarlo, se borra?¿Por qué lo que se hace es borrar primero la pantalla cuando no hay nada y luego escribir en ella? Bueno, tiempo al tiempo, pronto lo explicamos…

Es importante, ya que nos hemos topado con el primer "comando" de Bennu, aclarar el concepto de parámetro. Verás que cuando vayan apareciendo más y más comandos, TODOS llevarán al final de su nombre un par de paréntesis –()- y el consabido punto y coma. Dentro del paréntesis puede o no haber nada, o haber un número/letra/palabra/signo… (como en el caso de **delete_text()**), o haber dos números/letras/palabras/signos… separados por comas, o haber tres números/letras/palabras/signos… separados por comas, o haber cuatro,etc. Cada uno de los valores que aparecen dentro de los paréntesis se denominan parámetros de ese comando, y el número de ellos y su tipo (si son números, letras, palabras,etc) depende de cada comando en particular. ¿Y para qué sirven los parámetros? Para modificar el comportamiento del comando en algún aspecto.

Me explico: los comandos que no tienen parámetros son los más fáciles, porque hacen una cosa, su tarea encomendada, y punto. No hay posibilidad de modificación: siempre harán lo mismo de la misma manera. Cuando un comando tiene uno o más parámetros, también hará su función preestablecida, pero el cómo la hará viene dado por el valor de cada uno de los parámetros que tenga, los cuales modificarán alguna característica concreta de esa acción a realizar.

Por ejemplo: supongamos que tenemos un comando, el comando "lalala" que tiene –que "recibe", técnicamente dicho- un parámetro. Si el parámetro vale 0 lo que hará el comando será imprimir por pantalla "Hola, amigo"; si el parámetro vale 1 lo que hará el comando será imprimir por pantalla "¿Qué tal estás?" y si el parámetro vale 2 imprimirá "Muy bien",etc. Evidentemente, en las tres posibilidades el comando "lalala" hace esencialmente lo mismo: imprimir por pantalla (para eso es un comando, sino podríamos estar hablando de tres comandos diferentes), pero dependiendo del valor de su único parámetro –numérico, por cierto: no valdría en este caso dar otro tipo de valor (como una letra) porque si no seguramente daría error- su acción de imprimir por pantalla un mensaje es modificada en cierta manera. Así,

Lalala(0); -> Imprime "Hola amigo"
Lalala(1); -> Imprime "¿Qué tal estás?"
Lalala(2); -> Imprime "Muy bien"

Para entendernos: los comandos serían como los verbos, que hacen cosas, que ejecutan órdenes, y los parámetros serían como los adverbios, que dicen cómo se han de hacer estas cosas, de qué manera se ejecutan esas

órdenes (más lento, más suave, en la esquina superior o inferior de la pantalla, etc).

Dicho esto, vemos claramente que **delete_text()** recibe un único parámetro que ha de ser numérico. Y ¿qué significa en concreto que ese parámetro valga 0? Pues significa lo que hemos dicho ya: que **delete_text()** se dedicará a borrar TODOS los textos que aparezcan por pantalla. Poniendo otros valores como parámetros –ya lo veremos-, se podría especificar más y decirle a **delete_text()** que borrara tal o cual texto concreto.

En vez de poner como valor del parámetro de **delete_text()** el número 0, podríamos, de forma completamente equivalente escribir la palabra "**ALL_TEXT**" (con o sin comillas, no importa). Significa lo mismo. De hecho, si observaras la salida que genera la utilidad moddesc cuando se le consulta sobre el módulo "mod_text", podrías comprobar como "**ALL_TEXT**" es una constante (ya hablaremos de ellas en el capítulo 4) definida en dicho módulo, cuyo valor precisamente es igual a 0.

## *Línea "mivar1=mivar1+2;". Asignaciones, igualdades e incrementos

He aquí una línea que puede sorprender a quien no haya programado nunca: una "igualdad" que no es igual…¿Cómo es eso? Pues porque esto no es una igualdad: es una asignación. El valor de mivar1+2 se le asigna como nuevo valor a mivar1. Ojo: las asignaciones siempre hay que leerlas de derecha a izquierda. Expliquemos un poco mejor todo esto.

Hemos comentado que las variables son trocitos de memoria del ordenador con un nombre donde se guardan valores usados por el programa. Pues bien, lo que hace esta línea es coger el valor que tiene mivar1, sumarle dos, y el resultado guardarlo en mivar1 otra vez (con lo que perderíamos su antiguo valor). Es como abrir un cajón –la variable- y coger su contenido, sumarle dos, y posteriormente poner ese nuevo contenido sumado en el mismo cajón de antes (podría haber sido otro cajón, también).

De hecho, en Bennu, cuando queramos especificar que algo ES IGUAL a algo, no usaremos el signo = -signo de "asignación" -, sino que se usará el signo == (dos iguales) –signo de "igualdad". Cuidado porque son signos diferentes con significado diferentes y pueden dar muchos problemas para el que se despista. Un ejemplo de utilización del signo == sería por ejemplo en un while:

```
While (mivar1 == 30)
…
End
```

En este ejemplo, lo que se estaría comprobando es si mivar ES IGUAL a 30, y mientras eso sea verdad, realizar el bucle. Hubiera sido un gran error haber puesto

```
While (mivar1 = 30)
…
End
```

ya que la condición del while en este caso SIEMPRE sería verdad, porque lo que estaría pasando es que se le asignaría el valor 30 a la variable mivar1 (y eso siempre es verdad), y por lo tanto, nunca se incumpliría una condición que de hecho no es tal.

Por cierto, seguramente cuando empieces a leer códigos de otros desarrolladores te percatarás de que utilizan signos (o hablando más técnicamente, "operadores") un tanto peculiares. Estos operadores son del tipo ++,--, +=,-=,*=,/=, etc. No son imprescindibles, porque lo que hacen se puede hacer con los operadores aritméticos de toda la vida (+,-,*,/, etc) pero se utilizan mucho. En este manual no los utilizaremos por razones pedagógicas para facilitar así aún más la comprensión de los códigos fuente a aquellos que se inician en el mundo de la programación; sin embargo, es probable que cuando cojas un poco más de soltura empieces a usarlos porque facilitan mucho la escritura (y lectura) del código y para el programador son muy cómodos de utilizar.Pero, ¿qué es lo que hacen? A continuación presento una tabla de "equivalencias" –incompleta- entre el uso de algunos de los operadores "raros" de los que estamos comentando y su correspondencia con el uso de los operadores "normales" equivalente. Así pues, las expresiones:

| | | |
|---|---|---|
| a++ | Equivale a | a=a+1 .(Al operador "++" se le llama operador "incremento") |

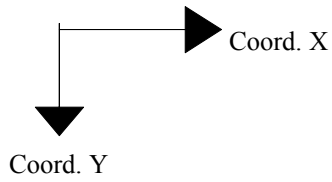
| | | |
|---|---|---|
| a-- | Equivale a | a=a-1 (Al operador "--" se le llama operador "decremento") |
| a+=3 | Equivale a | a=a+3 |
| a-=3 | Equivale a | a=a-3 |
| a*=3 | Equivale a | a=a*3 |
| a/=3 | Equivale a | a=a/3 |

## *Orden "Write()"

Se puede observar que este nuevo comando tiene bastantes parámetros. Su tarea principal es, como su nombre indica, escribir un texto determinado en pantalla. Pero,¿qué texto? ¿En qué sitio de la pantalla?...eso lo dirán el valor de sus parámetros.

No obstante, antes de saber cómo funcionan dichos parámetros, es necesario aprender algo primero: el origen de coordenadas en Bennu está en la esquina superior izquierda de la ventana donde se visualiza el videojuego (y si el videojuego se ve a pantalla completa, pues será la esquina superior izquierda de la pantalla).Ese punto es el punto (0,0). A partir del punto (0,0), si nos movemos para la derecha estaremos incrementando la coordenada X, y si nos movemos para abajo estaremos incrementando la coordenada Y. Un esquema sería:

(0,0): extremo superior izquierdo de la ventana madre




Ahora sí, el significado de los parámetros de la orden **write()** son:

| | | |
|---|---|---|
| 1r parámetro | Tipo entero | Indica un código correspondiente a la fuente de letra que se va a utilizar para el texto. Si no queremos usar ninguna fuente concreta, la fuente predeterminada del sistema se representa por el código 0. |
| 2º parámetro | Tipo entero | Coordenada horizontal en píxeles (respecto el origen de coordenadas: la esquina superior izquierda de la ventana del programa) del punto especificado en el 4º parámetro |
| 3r parámetro | Tipo entero | Coordenada vertical en píxeles (respecto el origen de coordenadas: la esquina superior izquierda de la ventana del programa) del punto especificado en el 4º parámetro |
| 4º parámetro | Tipo entero | Código correspondiente al tipo de alineación. Indica qué punto dentro del texto será el que se posicione exactamente en las coordenadas especificadas en el 2º y 3r parámetro.<br>*Si es igual a 0, la coordenada escrita en el 2º y 3r parámetro corresponderá al extremo superior izquierdo del texto.<br>*Si es igual a 1, la coordenada corresponderá al punto centrado horizontalmente en el texto de su límite superior.<br>*Si es igual a 2, la coordenada corresponderá al extremo superior derecho<br>*Si es igual a 3, corresponderá al punto centrado verticalmente del extremo izquierdo<br>*Si es igual a 4, el texto estará centrado horizontalmente y verticalmente alrededor de ese punto<br>*Si es igual a 5, corresponderá al punto centrado verticalmente del extremo derecho<br>*Si es igual a 6, la coordenada corresponderá al extremo inferior izquierdo<br>*Si es igual a 7 el punto estará centrado horizontalmente en el extremo inferior<br>*Si es igual a 8, la coordenada corresponderá al extremo inferior derecho. |

<table>
<tr><td></td><td></td><td>Tal vez lo más rápido sea jugar con la función cambiando el valor de este parámetro y ver los resultados. Aunque, de forma alternativa, en vez de asignar a este parámetro el valor numérico directamente, se pueden utilizar palabras concretas (que no son más otra vez que constantes definidas dentro del módulo “mod_text”) que significan lo mismo y son más entendibles a primera vista. Pueden ir entrecomilladas o no. Su tabla de equivalencias es la siguiente:<br><br>
<table>
<tr><th>Valor real del 4º parámetro</th><th>Valor posible a utilizar como alternativa</th></tr>
<tr><td>0</td><td>ALIGN_TOP_LEFT</td></tr>
<tr><td>1</td><td>ALIGN_TOP</td></tr>
<tr><td>2</td><td>ALIGN_TOP_RIGHT</td></tr>
<tr><td>3</td><td>ALIGN_CENTER_LEFT</td></tr>
<tr><td>4</td><td>ALIGN_CENTER</td></tr>
<tr><td>5</td><td>ALIGN_CENTER_RIGHT</td></tr>
<tr><td>6</td><td>ALIGN_BOTTOM_LEFT</td></tr>
<tr><td>7</td><td>ALIGN_BOTTOM</td></tr>
<tr><td>8</td><td>ALIGN_BOTTOM_RIGHT</td></tr>
</table>
</td></tr>
<tr><td>5º parámetro</td><td>Tipo cadena</td><td>El texto a escribir (entre comillas siempre).</td></tr>
</table>

Queda claro pues que la línea *write(0,mivar1,100,1,"¡Hola mundo!");* lo que hace, primero de todo, es escribir el texto “¡Hola mundo!” , con la fuente predeterminada del sistema (todavía no sabemos manejar con Bennu los diferentes tipos de fuente disponibles), y con la alineación de tipo 1, que quiere decir que la coordenada horizontal especificada en **write()** estará en el centro del texto y la coordenada vertical será el “techo” justo debajo del cual se escribirá el mensaje.

La coordenada vertical es fácil de ver: vale 100 píxeles. Si valiera más veríamos como el texto aparece más abajo en la ventana, y si vale menos, pues saldría más arriba; con lo dicho tendrías que deducir cómo saldría el texto con el tercer parámetro igual a 0. Pero la coordenada horizontal es un poco más difícil: vale *mivar1*. Y claro, *mivar1* vale un número, ya lo sabemos, pero ¿cuál?

Acabemos primero, no obstante, de explicar la última orden, y enseguida veremos cómo podemos averiguar el valor del 2º parámetro del **write()**. *(*De hecho, ya deberías de ser capaz de adivinarlo con lo explicado).

## *Orden "Frame"

Este comando es muy especial y FUNDAMENTAL: en TODOS los programas Bennu aparece, y fíjate que es de los pocos que no llevan paréntesis alguno. Tampoco es necesario importar ningún módulo para poderlo usar: es una palabra reservada del lenguaje.

Antes de nada, prueba de ejecutar el mismo programa sin esta orden (bórrala del código fuente, o mejor, coméntala detrás de las // :así el compilador no la leerá pero bastará retirar otra vez las // para tenerla otra vez disponible). Verás que el programa no parece ejecutarse bien. Vaya.

Imagínate que el comando Frame es como si fuera una puerta, un agujero o algo parecido que permite visualizar por pantalla todo lo que el programa ha estado ejecutando hasta el momento durante un instante minúsculo. Si no hay Frame, por mucho que el programa haga, no se verá nada porque esa puerta, ese agujero estará cerrado. Cada vez que se lea el comando Frame, ¡zas!, se abre la puerta, se visualiza todo lo que en todas las líneas anteriores (desde el Frame anterior) ha estado haciendo el programa y se vuelve a cerrar la puerta, así hasta el siguiente Frame.

De esta manera, un programa Bennu se basa en trozos de código que se van ejecutando pero que

realmente no se ven sus resultados en pantalla hasta que llega un Frame. A partir de él, si hay más líneas de código se vuelven a ejecutar, pero sin llegar a visualizarse hasta que no se encuentre con el siguiente Frame. Y así.

Es por eso, y esto es muy importante que lo recuerdes, TODOS los programas en Bennu contienen un bucle principal donde en su interior aparece un Frame al final. Lo que hay en el bucle es el programa propiamente dicho que se va ejecutando hasta que se llega al Frame, que visualiza el resultado, y luego se va a la siguiente iteración, donde el programa vuelve a ejecutarse y se vuelve a llegar al Frame, que vuelve a visualizar los resultados –que serán diferentes de la primera vez dependiendo del valor que vayan cogiendo las distintas variables, etc-, y se vuelve a ir a la siguiente iteración, y así, y así se consigue un programa cuya ejecución será visible durante un tiempo: el tiempo que se tarde en salir de ese bucle principal (bien porque es un bucle finito por una condición –como el while del ejemplo-, o bien porque se sale forzadamente de un bucle infinito).

Ésta es una explicación más o menos de estar por casa: posteriormente volveremos a explicar más profundamente el sentido de este comando (cuando hablemos de los procesos), pero por ahora esto es lo que debes entender.

De hecho, si has entendido bien el funcionamiento de Frame, podrás entonces contestar por qué hemos puesto primero **delete_text()** cuando no había ningún texto que borrar y luego hemos puesto el **write()**. Acabamos de decir que el Frame se encarga de poner en pantalla el resultado de todas las operaciones que ha venido efectuando el programa hasta ese momento (lo que se dice el "estado" del programa).Si ponemos el **write()** primero, y luego el **delete_text()**, primero escribiremos algo y luego lo borraremos, con lo que cuando se llegue al Frame en ese momento no habrá ningún texto que visualizar en pantalla y se verá una pantalla negra y ya está. Pruébalo. Podrás comprender pues la importancia de saber colocar en el lugar apropiado de tu código el Frame: será ese lugar donde estés seguro de que quieres mostrar el estado de tu programa. Generalmente el Frame es la última línea del bucle principal del programa.

## *Funcionamiento global del programa

Ahora que hemos acabado de explicar las líneas una a una, podríamos ver cuál es el funcionamiento global del programa. Primero se le da el valor de 10 a *mivar1*. Luego se entra en lo que es el bucle principal –ineludible en Bennu- del programa. Es un while que se ejecutará siempre que *mivar1* sea menor que 320. Evidentemente, cuando se llega aquí por primera vez esto es cierto porque acabamos de darle el valor de 10 a *mivar1*; por lo tanto, entramos en el bucle. Borramos todos los textos que hubiera en pantalla, aumentamos el valor de *mivar1* en 2 –con lo que en esta primera iteración ahora *mivar1* vale 12- y escribimos un texto en unas coordenadas por (*mivar1*,100), que en esta primera iteración serán (12,100). Finalmente llegamos a Frame, que lo que hará es imprimir en pantalla el resultado de nuestro programa, que no deja de ser poner un texto en las coordenadas (12,100). Segunda iteración: borramos todos los textos que hubiera en pantalla aumentamos *mivar1* en 2 –ahora valdrá 14-, y escribimos el mismo texto en –ahora- (14,100), con lo que al llegar al Frame, ahora en la segunda iteración el texto aparece un poco más a la derecha que la primera vez. Tercera iteración: mismo proceso pero ahora se escribe el texto en las coordenaas (16,100).Y así. ¿Y así hasta cuando? Pues hasta que *mivar1*, que define la coordenada horizontal del texto, valga 320.

¿Y por qué es 320? Fíjate que da la casualidad que *mivar1* vale 320 justo cuando el texto (o mejor dicho, según el 4º parámetro del **write()**, el punto horizontal central del texto) ha llegado al otro extremo de la ventana. Por tanto, podemos concluir que el extremo derecho de la ventana está en x=320. De hecho, date cuenta que nosotros en ningún momento hemos establecido el tamaño de la ventana de nuestro juego; esto se puede hacer, pero si no se hace, por defecto Bennu crea una ventana de anchura 320 píxels y de altura 240 píxels, por lo que sabremos siempre que la coordenada horizontal de los puntos del extremo derecho de la ventana valdrá 320.¿Qué pasaría si en el while, en vez de poner 320 ponemos por ejemplo 3200? Pues que el texto se movería igual hasta llegar al extremo derecho de la ventana (x=320), pero como el bucle del while todavía no habría acabado, aunque el texto habría desaparecido por la derecha y ya no sería visible, el programa continuaría ejecutándose –nosotros veríamos sólo una ventana negra- hasta que por fin, después de un rato, *mivar1* llegara a valer 3200, momento en el que el programa acabaría. ¿Y si hiciéramos al revés, en vez de poner en el while 320 ponemos 32? Pues ocurriría lo contrario: el programa acabaría antes de que el texto pudiera llegar al extremo derecho, de hecho acabaría enseguida porque el texto pronto llega a esa coordenada.

También nos podemos preguntar otras cosas. ¿Qué pasaría si, en el programa original, asignáramos un valor inicial de 100 a *mivar1*? Pues, evidentemente, que al iniciar el programa el texto ya aparecería en la ventana en una posición más centrada horizontalmente, y tendría menos recorrido para llegar hasta la coordenada x=320, con lo

que el programa también duraría menos. ¿Y si inicializáramos *mivar1* a 1000? Pues que no veríamos nada de nada porque nada más empezar la condición del while resulta falsa, y como tal, el while es saltado y se acaba el programa. También podemos probar de inicializar *mivar1* a 1000 y además cambiar la condición del while para que *mivar1*<3200. Es este caso, el programa se ejecutaría un rato, hasta que *mivar1* llegara a valer 3200, pero no veríamos nada tampoco, porque el valor inicial de *mivar1* ya de entrada es muy superior a la coordenada derecha de la ventana del juego, y por tanto, el texto estará fuera de la región visible.

Otro efecto curioso es el de cambiar el incremento en el valor de la variable *mivar1*. Así, en vez de hacer *mivar1=mivar1+2;* podríamos haber escrito *mivar1=mivar1+10;*, por ejemplo. En este caso, veríamos como el texto se mueve mucho más rápido, ya que en cada iteración, el texto se desplaza una cantidad mayor de píxels y por tanto recorre la ventana en menos iteraciones.

Podemos jugar con **write()** también. Si ponemos un valor constante en su segundo parámetro, es evidente que el texto no se moverá, pero aún así, veremos que el programa tarda un rato en cerrarse. Esto es, evidentemente, porque aunque no hayamos utilizado *mivar1* para hacer el scroll del texto, el bucle del while sigue funcionando, y cuando se llega a la condición falsa se sale igual: estaremos borrando y escribiendo en cada iteración una y otra vez el mismo texto en la misma posición hasta que la variable *mivar1* valga 320. También podríamos probar de poner *mivar1* en el tercer parámetro del **write()**; de esta manera, haríamos un scroll vertical del texto. En este caso, el extremo inferior de la pantalla es 240, así que si no queremos esperar un rato con la pantalla negra después de que el texto haya desaparecido por debajo de la ventana, tendríamos que modificar la condición del while consecuentemente. Y podríamos incluso hacer un scroll en diagonal, simplemente añadiendo otra variable y poniendo las dos variables en el 2º y 3r parámetro del **write()**. Algo así como esto:

```
Import "mod_text";

Process Main()
Private
          int mivar1;
          int mivar2;
End
Begin
          mivar1=10;
           mivar2=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    mivar2=mivar2+2;
                    write(0,mivar1,mivar2,1,"¡Hola mundo!");
                    frame;
          end
end
```

Fíjate que lo único que hemos hecho es declarar, inicializar e incrementar correspondientemente una nueva variable que representará la coordenada vertical donde se escribirá el texto. La condición del while puede continuar siendo la misma o depender de la nueva variable, según nos interese.

Deberías intuir ya con lo explicado hasta ahora cómo se podría hacer, por ejemplo, un scroll vertical que fuera de abajo a arriba. El código sería éste:

```
Import "mod_text";

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=240;
          while(mivar1>10)
                    delete_text(0);
                    mivar1=mivar1-2;
                    write(0,100,mivar1,1,"¡Hola mundo!");
                    frame;
          end
```

```
end
```

Lo que se ha hecho ha sido: primero, darle un valor inicial a *mivar1* de 240; segundo, cambiar la condición del while poniendo que el bucle se acabará cuando *mivar1* deje de ser mayor que 10; y tercero, el incremento de la *mivar1* lo hago negativo. Con estos tres cambios, lo que consigo es que inicialmente el texto se escriba en el extremo inferior de la ventana, y que en cada iteración se vaya escribiendo 2 píxeles más arriba –recordad que la coordenada vertical decrece si vamos subiendo por la ventana- y este proceso se irá ejecutando hasta que lleguemos al momento donde *mivar1* vale menos de 10, es decir, que el texto se ha escrito casi en el extremo superior de la ventana.

Otro efecto interesante es el de comentar la línea **delete_text()**. Al hacer esto, estamos evitando que en cada iteración el texto acabado de escribir se borre, con lo que estaremos escribiendo cada vez un nuevo texto sobre el anterior, desplazado unos píxeles a la derecha. El efecto de superposición que se crea es un línea contínua curiosa.

Y otra cosa que puedes probar es (con el **delete_text()** descomentado) cambiar el valor de su parámetro -0- por otro número cualquiera. La explicación de lo ocurrido lo dejaremos para más adelante.

## *Notas finales

Fíjate en las tabulaciones: he tabulado todo el código entre el BEGIN/END y luego, otra vez, he vuelto a tabular el código entre el WHILE/END. Esto, ya lo comenté anteriormente, es por claridad. Tabulando se tiene una rápida visión de dónde empiezan y dónde acaban los distintos trozos de código que componen nuestro programa, y podemos localizar rápidamente zonas concretas. Tabulando sabemos que un conjunto de sentencias pertenecen a un mismo bucle, condicional, programa, etc, aportando así una gran claridad en la lectura del código. Y sobretodo, ayudando en gran medida para corregir posibles errores de escritura. Es muy recomendable acostumbrarse a tabular los distintos tipos de bucles, condiciones, procesos,etc, para una buena salud mental y ocular. Ya iréis viendo ejemplos de programas donde se pone en práctica esta manera de organizarse (por otra parte, estándar en el mundo de la programación). Al final de este capítulo puedes consultar un extracto de la "Guía de Estilo", documento que propone una serie de pautas sobre éstos y otros detalles que pretenden facilitar la lectura y la escritura de código Bennu, estandarizando convenios de codificación.

Otra cosa que posiblemente no hayas visto es cuándo en general se escriben los puntos y coma y cuándo no. En general, si no digo explícitamente lo contrario, los puntos y coma hay que escribirlos SIEMPRE al final de una sentencia (como puede ser un comando, una asignación, o cualquier cosa). Prácticamente la única excepción –que de hecho no es tal porque no son sentencias- es cuando abrimos y cerramos un bloque de código, como puede ser un bucle (WHILE/END es un ejemplo) , un condicional (IF/END…) o el bloque principal del programa (BEGIN/END). En general, todas las parejas de palabras que tengan la palabra END no llevan punto y coma. Y la línea que comienza con la palabra PROCESS,tampoco.

Y un último comentario. Te habrás fijado que, independientemente del trozo de bloque que cerremos (bucle, condicional, begin principal…), todos acaban con la misma palabra: END. Esto a la larga puede ser un poco lioso porque cuando tienes cuatro o cinco ENDs seguidos no sabes cuál corresponde a qué bloque. Por eso siempre es muy recomendable añadir un comentario al lado del END diciendo a qué bloque corresponde, de esta manera:

```
Import "mod_text";

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end //End del while
```

```
end //End del begin
```

Así no tendrás tantas posibilidades de confusión.

## Explicación paso a paso de "Mi segundo programa en Bennu"

Recordemos el código:

```
Import "mod_text";
Import "mod_rand";

Process Main()
private
          int mivar1;
end

begin
          loop
                    delete_text(0);
                    mivar1=rand(1,10);
                    if (mivar1<=5)
                              write(0,200,100,2,"Menor o igual que 5");
                    else
                              write(0,200,100,2,"Mayor que 5");
                    end
                    frame;
          end
end
```

De aquí, lo único novedoso son dos cosas. Por un lado, importamos un nuevo módulo, además del ya conocido "mod_text": el módulo "mod_rand". Esto es así porque en el código hemos utilizado un comando incluido en dicho módulo (como podemos comprobar mediante la utilidad *moddesc*), el comando **rand()** .Lo segundo que es novedoso es lo que hay entre el BEGIN/END, o sea:

```
loop
          delete_text(0);
          mivar1=rand(1,10);
          if (mivar1<=5)
                    write(0,200,100,2,"Menor o igual que 5");
          else
                    write(0,200,100,2,"Mayor que 5");
          end
          frame;
end
```

Podemos ver que esto es un bloque definido en el inicio por la palabra LOOP y terminado al final por –siempre- la palabra END. Como bloque que es, su interior ha sido tabulado para facilitar la lectura

## *Bloque "Loop/End". Sentencias BREAK y CONTINUE. Orden "Write_var()".

La sentencia LOOP (bucle) es una sentencia que implementa un bucle infinito, es decir, que repite indefinidamente un grupo de sentencias. Para implementar este bucle vemos que se debe comenzar con la palabra reservada LOOP, seguida de las sentencias que se quieren repetir continuamente y la palabra reservada END al final. Cuando un programa se encuentra una sentencia LOOP...END se ejecutarán a partir de entonces, una y otra vez, todas las sentencias interiores a dicho bucle. Las sentencias interiores a un bucle LOOP pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles LOOP.

Aprovecho aquí para comentar un par de órdenes muy ligadas al bucle LOOP y a todos los bucles en general, aunque no aparezcan en el ejemplo

| ***Sentencia BREAK;*** |
|---|
| *Si en algún momento quisiéramos finalizar un bucle LOOP, se puede utilizar la sentencia BREAK; que, al ejecutarse dentro de un bucle de este tipo, forzará al programa a seguir a continuación de su END. Es decir, una sentencia BREAK; dentro de un bucle lo finalizará de forma inmediata, continuando el programa por la sentencia siguiente a dicho bucle.<br><br>*Esta sentencia puede ponerse, además de dentro del bucle LOOP, dentro de los bucles WHILE, REPEAT, FOR o FROM.(ya los veremos). Por ejemplo, una sentencia BREAK dentro de un bucle WHILE funcionará de la misma manera: finalizará el bucle de forma inmediata, continuando el programa a partir de la sentencia siguiente a dicho bucle.<br><br>*En caso de haber varios bucles anidados (unos dentro de otros) la sentencia BREAK saldrá únicamente del bucle más interior de ellos.<br><br>*La sentencia BREAK no es válida para finalizar otras secuencias que no sean bucles, como IF o SWITCH, ni sentencias CLONE (ya las veremos) |
| ***Sentencia CONTINUE;*** |
| *La sentencia CONTINUE;, dentro de un bucle, finalizará la iteración actual y comenzará la siguiente (es decir: el programa continuará ejecutándose justo tras la palabra LOOP otra vez). Es decir, una sentencia CONTINUE; dentro de un bucle forzará al programa a finalizar la iteración actual del mismo y comenzar la evaluación de la siguiente.<br><br>*Aparte del bucle LOOP,puede usarse esta sentencia en los bucles WHILE,REPEAT, FOR o FROM (ya los veremos). Por ejemplo, una sentencia CONTINUE dentro de un bucle WHILE forzará al programa a comprobar la condición inicial inmediatamente, y si ésta es cierta, volverá a ejecutar las sentencias interiores desde el principio (tras el WHILE), y si la condición resultara falsa, la sentencia CONTINUE finalizará el bucle.<br><br>*En caso de haber varios bucles anidados (unos dentro de otros) la sentencia CONTINUE tendrá efecto únicamente en el bucle más interior de ellos.<br><br>*La sentencia CONTINUE no es válida para finalizar otras secuencias que no sean bucles, como IF o SWITCH, ni sentencias CLONE (ya las veremos). |

Vamos a ver estas sentencias con ejemplos. Si tenemos este código:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
          Loop
                    mivar1=mivar1+1;
                    If (mivar1==30)
                              Break;
                    End
                    write_var(0,100,100,4,mivar1);
                    Frame;
          End
End
```

lo que tendrá que ocurrir es que aparecerá escrito en pantalla un número que irá aumentando en una unidad desde el 0 hasta el número 29.¿Por qué? Porque inicializamos la variable *mivar1* con el valor 0, y en cada iteración le incrementamos en una unidad su valor y lo escribimos. Esto sería un proceso infinito si no fuera porque en cada iteración se comprueba una condición –el "if" lo veremos enseguida, pero aquí es fácil ver cómo funciona- que dice que si *mivar1* en ese momento vale exactamente 30, entonces se haga un break. Y hacer un break implica salir

automáticamente del interior del bucle donde se está en este momento. Como en este ejemplo, al salir del bucle ya no hay nada más, se acaba el programa. Hay que puntualizar que sólo se verá hasta el número 29 porque la condición de comprobación está antes del **write()**, y por tanto, cuando *mivar1* sea igual a 30, se saldrá del bucle sin pasar por el **write()**, y por tanto, el valor 30 no se escribirá. Otra cosa sería –es fácil verlo-, si hubiéramos escrito el mismo programa pero cambiando el orden:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
          Loop
                    mivar1=mivar1+1;
                    write_var(0,100,100,4,mivar1);
                    If (mivar1==30)
                              Break;
                    End
                    Frame;
          End
End
```

Un par de detalles: fijarse que para escribir el valor de una variable no hemos utilizado el comando **write()** sino otro muy parecido que es **write_var()**. Este comando funciona igual que **write()**, pero este último sólo sirve para escribir textos concretos y fijos establecidos como 5º parámetro entre comillas, y en cambio **write_var()** sirve para escribir el valor que tiene una variable, pasada también como 5º parámetro (sin comillas).

Más correctamente, la gran diferencia entre **write()** y **write_var()** es que el valor que se escriba con **write()** quedará permanentemente inalterado en pantalla una vez escrito, y lo que se escriba con **write_var()** (valores de variables), automáticamente en cada frame se comprobará si ha cambiado; en caso afirmativo ese dato se refrescará en pantalla mostrando su nuevo valor. Por tanto, la función **write_var()** es la más adecuada para hacer un seguimiento "en tiempo real" frame a frame de la evolución del valor de una variable.

Otra cosa a tener en cuenta es, como siempre, recordar la existencia de la orden "frame;" al final del bucle principal del programa, ya que si no está no se ve nada, tal como hemos visto.

Cuando estudiemos el funcionamiento del "if" lo veremos mejor, pero fíjate ahora que en la condición de igualdad *if(mivar1==30)* he escrito dos iguales seguidos (==) y no uno. Esto es porque ya hemos comentado que el signo = a secas es un operador de asignación, pero no de igualdad. Para escribir que a es igual a b, hemos de escribir a==b; si escribiéramos a=b estaríamos diciendo que el valor de b pasa a ser el valor de a, y por tanto, una sentencia como if (a=b) siempre sería verdad, porque siempre sería verdad, si no hay errores, que le paso el valor de b a a.

Ahora pongo otro ejemplo para ver el funcionamiento del CONTINUE:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
          Loop
                    mivar1=mivar1+1;
                    If (mivar1==30)
                              continue;
                    End
                    write_var(0,100,100,4,mivar1);
                    Frame;
          End
End
```

Es el mismo código que antes, pero cambiando BREAK por CONTINUE. Lo que ocurre, en cambio, es que se genera un bucle infinito con el número que aparece en pantalla incrementando su valor en una unidad hasta el

infinito (de hecho en teoría hasta su valor máximo que puede albergar según el tipo de variable que lo contiene, pero ocurre un error antes que no nos incumbe ahora). Entonces, ¿qué hace el CONTINUE? El programa se ejecuta muy rápido para verlo, pero lo que ocurre es que el número 30 no aparece: se visualiza hasta el 29 y después se ve el 31 y siguientes. Lo que ocurre es que cuando se realiza la comprobación del if y se ve que en esa iteración *mivar1* vale 30, se salta todo lo que queda de iteración y se vuelve otra vez para arriba, a la palabra LOOP, y se empieza la siguiente iteración. Es decir, hacer un CONTINUE es dejar inutilizadas las sentencias que hay dentro del bucle posteriores a él en esa iteración concreta. Así pues, cuando *mivar1* vale 30, no se hace el **write_var()** y se vuelve al LOOP, donde *mivar1* pasa a valer 31.

En este caso, es indiferente poner el if antes o después del **write_var()**, porque en ambos casos la sentencia Frame es la que se salta si se ejecuta el CONTINUE. Si se salta la sentencia frame, nada se muestra por pantalla.

## *"mivar1=rand(1,10);". Valores de retorno de las funciones

Esta línea es una simple asignación de un valor a una variable, pero en vez de ser un valor concreto como podría ser *mivar1=10;,* por ejemplo, lo que se hace es utilizar el comando **rand()**.

Ves que el comando **rand()** que es una orden con dos parámetros. Lo que no hemos comentado hasta ahora es que los comandos, a parte de recibir parámetros de entrada –si lo hacen-, y a parte de hacer la tarea que tienen que hacer, también devuelven valores de salida, generalmente INT. Estos valores enteros que devuelven pueden servir para varias cosas dependiendo del comando; el significado de ese valor devuelto depende: algunos valores son de control, avisando al ordenador que el comando se ha ejecutado bien o mal, otros sirven para generar números específicos, (como **rand()**), etc.

De hecho, no lo hemos visto pero el comando **write()** por ejemplo devuelve un valor entero cada vez que se ejecuta. Si nos interesa recoger ese valor, se lo asignamos a una variable y lo utilizamos, y si no, pues no hacemos nada. Ahora no nos interesa saber cual es el significado de este número que devuelve **write()**, pero para que lo veas, ejecuta esto:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
          Loop
                    mivar1=write(0,100,100,4,"Hola");
                    write_var(0,150,150,4,mivar1);
                    Frame;
          End
End
```

Verás que se escribe la palabra "Hola" y que además aparece un número que va aumentando hasta que deja de hacerlo -momento en el cual se ha producido el mismo error que con el ejemplo del CONTINUE, que ya estudiaremos-. Lo que nos interesa ahora es averiguar qué significa este número que aparece en pantalla. Podrás intuir que es el valor que en cada iteración devuelve el comando **write()**. Efectivamente: en este ejemplo, cada vez que se llama a **write()**, éste realiza su conocida tarea de imprimir por pantalla la palabra "Hola",pero además, recogemos el valor entero que devuelve –cosa que antes no hacíamos porque no nos interesaba, aunque **write()** lo devuelve siempre- en una variable, para seguidamente visualizar dicho valor también por pantalla.

En el caso de **write()**, el valor devuelto representa un número identificador del texto impreso en pantalla; es como un DNI de la frase escrita,un número al que podremos hacer referencia en nuestro programa posteriormente y mediante el cual podremos manipular ese texto cuando lo deseemos. Es decir, que si escribimos un texto con **write()** y recogemos lo que devuelve, allí tendremos el número que identificará ese texto, y si luego en nuestro programa queremos cambiarlo o hacer lo que sea con él, simplemente tendremos que recurrir a ese número para saber de qué texto estamos hablando y hacer lo que sea con él (ya veremos ejemplos, tranquilo).

El error que hemos visto que ocurre cuando el programa ya lleva un tiempo ejecutándose es debido a que se están escribiendo demasiados textos en pantalla. ¡Cómo, si sólo escribo "Hola" y ya está!. Pues no, fíjate bien. Si acabo de decir que lo que devuelve el **write()** es un identificador del texto escrito, y ese identificador vemos que sube como la espuma, eso quiere decir que estamos escribiendo muchos textos diferentes, cada uno de los cuales, como es lógico, tiene un identificador diferente. Es decir, que si vemos que el número que aparece sube hasta el 500, es porque hemos escrito 500 textos diferentes, cada uno de los cuales con un identificador diferente. ¿Y esto cómo es? Pues porque en cada iteración ejecutamos la orden **write()** de nuevo, por lo que en cada iteración estamos escribiendo un nuevo texto, que se superpone al anterior. Es por esta razón que al final aparece este error,porque hemos escrito tantos textos en pantalla que la memoria del ordenador se desborda. Entonces, ¿cómo se puede escribir un texto fijo que esté presente durante toda la ejecución del programa sin que dé el error? Prueba esto:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
           mivar1=write(0,100,100,4,"Hola");
           write_var(0,150,150,4,mivar1);
           Loop
                     Frame;
           End
End
```

Deberías entender lo que ocurre. Escribimos una sola vez "Hola", vemos como el identificador de ese texto, efectivamente es el número 1 (siempre se crean los identificadores a partir del 1 para el primer texto escrito y a partir de allí para arriba) y es sólo después cuando nos metemos en el bucle principal del programa con el frame, para que se pueda visualizar la frase fija todas las veces que haga falta.

Otra solución al mismo problema sería:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
           Loop
                     delete_text(0);
                      mivar1=write(0,100,100,4,"Hola");
                     write_var(0,150,150,4,mivar1);
                     Frame;
           End
End
```

Hemos hecho el mismo truco que en nuestro primer programa en Bennu.

Ahora es el momento donde se puede entender mejor cuál es el significado del parámetro del **delete_text()**. Ya sabemos que si ponemos un 0 quiere decir que borrará todos los textos que haya en pantalla –escritos con el comando **write()**-. Prueba de ejecutar el mismo programa anterior pero poniendo un 1 como valor del **delete_text()**. Y luego poniendo un 2,etc. Verás que se vuelven a escribir muchos textos otra vez y se vuelve a generar el error que hemos visto. ¿Por qué? Porque si no ponemos un 0 en **delete_text()** y ponemos cualquier otro número, ese número representará el identificador de un texto concreto, y sólo borrará ese texto concreto. Es decir,que si con **write()** hemos creado un texto con identificador 1, **delete_text(1)** sólo borrará ese texto y ninguno más. Por eso, con este código,por ejemplo:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
```

```
Begin
            Loop
                        delete_text(1);
                         mivar1=write(0,100,100,4,"Hola");
                        write_var(0,150,150,4,mivar1);
                        Frame;
            End
End
```

lo que ocurre es que **delete_text()** sólo borrará un texto (el que tenga identificador 1) de entre todos los textos que se escriban –que van a ser tantos como iteraciones haya: infinitos, y cada uno con un identificador diferente-. Por tanto, el error volverá a aparecer porque sólo hemos borrado el primer texto que se escribió, pero a partir del segundo (con identificador 2) para adelante no se ha borrado ninguno.

Bueno. Una vez explicado todo este rollo sobre los valores de retorno de las funciones (o comandos) de Bennu, vamos al ejemplo que teníamos entre manos. Teníamos la línea:

```
mivar1=rand(1,10);
```

Claramente, *mivar1* va a valer un número que será lo que devuelva el comando **rand()**. El comando **rand()** precisamente sirve para devolver un número, cuya característica más importante es que es pseudo-aleatorio.Este comando tiene dos parámetros enteros: el primero es el límite inferior y el segundo es el límite superior –ambos incluidos- de los valores que puede tomar el número que **rand()** va a generar. Cada llamada a esta función generará un número diferente, siempre dentro del rango especificado. Por lo tanto, cada vez que se ejecute la línea anterior, *mivar1* va a valer un número diferente, entre 1 y 10.

## *Bloque "If/Elif/Else/End". Condiciones posibles.

La sentencia IF no es ningún bucle. Como dice su nombre, simplemente sirve para ejecutar un bloque de sentencias solamente si se cumple una condición determinada. Opcionalmente, también es posible incluir una sección ELSE con otro bloque de sentencias que se ejecutarían en caso de que NO se cumpla esa condición, teniendo pues el programa respuesta para las dos posibilidades. Es decir, en general, la sintaxis del bloque IF/ELSE será:

```
IF (condición)
            Sentencias -una o más-que se ejecutan si la condición es cierta;
ELSE
            Sentencias –una o más- que se ejecutan si la condición es falsa;
END
```

Tanto si se ejecutan las sentencias de la sección IF como si se ejecutan las sentencias de la sección ELSE, cuando se llega a la última línea de esa sección (una u otra), se salta a la línea inmediatamente posterior al END para continuar el programa.

Hemos dicho que el ELSE es opcional. Si no ponemos ELSE, el IF quedaría:

```
IF (condición)
            Sentencias -una o más-que se ejecutan si la condición es cierta;
END
```

En este caso, si la condición fuera falsa, el if no se ejecuta y directamente se pasa a ejecutar la línea inmediatamente posterior al END (el programa no hace nada en particular cuando la condición es falsa).

Es posible anidar sentencias IF sin ningún límite, es decir, se pueden poner mas sentencias IF dentro de la parte que se ejecuta cuando se cumple la condición (parte IF) o dentro de la que se ejecuta cuando la condición no se cumple (parte ELSE).

También existe la posibilidad de incluir dentro del bloque condicional una o varias secciones ELIF

(siendo en este caso también opcional la sección ELSE final):

```
IF (condición)
        Sentencias -una o más-que se ejecutan si la condición es cierta;
ELIF (condición)
        Sentencias –una o más- que se ejecutan si la condición anterior era falsa y la actual es cierta;
ELIF (condición)
        Sentencias –una o más- que se ejecutan si las condiciónes anteriores eran falsas y la actual cierta;
...
ELSE
        Sentencias –una o más- que se ejecutan si todas las condiciones anteriores eran falsas;
END
```

Como se puede ver, la sección ELIF se tiene en cuenta siempre y cuando las condiciones avaluadas hasta entonces hayan sido falsas, y la condición del propio ELIF sea la cierta. Es decir, un bloque IF(condicion1)/ELIF(condicion2)/END se puede leer como "si ocurre *condicion1*, haz el interior del primer if, y si no, mira a ver si ocurre *condicion2*; (sólo) si es así, haz entonces el interior del elif.". Ésta es una manera de hacer comprobaciones de condiciones múltiples...aunque existe una forma un poco más "depurada" que es utilizando el bloque SWITCH/END (explicado más adelante)

Finalmente, ¿qué tipo de condiciones podemos escribir entre los paréntesis del IF? Cuando vimos el WHILE en el primer programa que hemos hecho, vimos los operadores para las condiciones de menor que (<) o mayor que (>) y el de la condición de igualdad (==), cuyo operador recordad que son dos signos iguales, ya que un solo signo representa el operador de asignación. Pero hay muchas más operadores para diferentes condiciones:

| | |
|---|---|
| = = | Comparación de igualdad |
| <> | Comparación de diferencia (también sirve !=) |
| > | Comparación de mayor que |
| >= | Comparación de mayor o igual que (también sirve =>) |
| < | Comparación de menor que |
| <= | Comparación de menor o igual que (también sirve =<) |

Además, también se pueden utilizar los operadores llamados lógicos, usados para encadenar dos o más comprobaciones dentro de una condición. Los operadores lógicos son:

| | |
|---|---|
| **OR** | Comprueba que, al menos, una de dos expresiones sea cierta. (también sirve \| \| ) |
| **AND** | Comprueba que las dos expresiones sean ciertas. (también sirve &&) |
| **XOR** | Comprueba que sólo una de las dos expresiones sea cierta (también sirve ^^ ) |
| **NOT** | Comprueba que no se cumpla la siguiente condición (también sirve ! ) |

Los paréntesis sirven dentro de una condición para establecer el orden de comparación de varias expresiones, pues siempre se comparan primero las expresiones situada dentro de los paréntesis, o en el más interior, si hay varios grupos de paréntesis anidados.

Ejemplos de condiciones podrían ser:

| | |
|---|---|
| *mivar1==100 AND mivar2>10* | Esta condición comprueba que *mivar1* sea exactamente 100 Y QUE ADEMÁS, *mivar2* sea mayor que 10 |
| *mivar1<>0 OR (mivar2>=100 and mivar2<=200)* | Esta condición primero comprueba lo que hay dentro del paréntesis, o sea, que *mivar2* sea mayor o igual que 100 Y QUE ADEMÁS *mivar2* sea menor o igual que 200. Eso, O que *mivar1* sea diferente de 0 |

El anidamiento de condiciones puede ser todo lo complejo que uno quiera, pero conviene asegurarse de

que realmente se está comprobando lo que uno quiere, cosa que a veces, con expresiones muy largas, es difícil.

No está de más finalmente tener presente un par de reglas matemáticas que cumplirán todas las posibles expresiones que podamos escribir:

| | | |
|---|---|---|
| !(condicion1 OR condicion2) | Equivale a | !condicion1 AND !condicion2 |
| !(condicion1 AND condicion2) | Equivale a | !condicion1 OR !condicion2 |

Un dato importante: a la hora de realizar comprobaciones de condiciones en cláusulas IF, por ejemplo, el resultado de la comprobación está claro que sólo puede ser o VERDADERO o FALSO. Pero estos valores lógicos un ordenador, y Bennu en concreto, no los entiende, porque sólo entienden de ceros y unos. Así que Bennu entiende internamente que si una condición es falsa en realidad él entiende que vale 0, y si es verdadera, en realidad él entiende que vale diferente de 0 (1,-1 o lo que sea). Esta asociación (FALSO=0 y VERDADERO!=0) es común a otros muchos lenguajes de programación (C, Java, Php, Python, etc), pero, por comodidad, en todos estos lenguajes de programación -incluido Bennu- existe la posibilidad de utilizar en vez de los valores numéricos, dos constantes lógicas: "TRUE" y "FALSE", que equivalen a éstos.

También hay que tener en cuenta, en relación al párrafo anterior, que es muy probable que te encuentres en bastantes códigos expresiones tales como *if(mivariable)* en vez de *if(mivariable!=0)* por ejemplo. Es decir, podrás ver ifs que sólo incluyen una variable,pero no la condición para evaluarla. Esta forma de escribir los ifs es simplemente un atajo: si el el if sólo aparece una variable, función o expresión sin ningún tipo de comparativa, es equivalente a comprobar si dicha variable es VERDADERA (es decir, si es diferente de 0). En el caso que en el if aparezca una función tan sólo, sería equivalente a comprobar si el valor que devuelve esa función es, también, diferente de 0.

Bien. Retomemos otra vez nuestro programa, que lo tenemos un poco olvidado. Teníamos, recordemos, el siguiente código:

```
Import "mod_text";

Process Main()
Private
          int mivar1=0;
End
Begin
           Loop
                     delete_text(0);
                     mivar1=rand(1,10);
                     if(mivar1<=5)
                               write(0,200,100,2,"Menor o igual que 5");
                     else
                               write(0,200,100,2,"Mayor que 5");
                     end
                     frame;
          End
End
```

Está claro que lo que hace el programa es, una vez que el comando **rand()** ha devuelto un número aleatorio entre 1 y 10 –ambos incluidos- y lo ha asignado a la variable *mivar1*, comprueba el valor de dicha variable con un if. Si el recién valor de *mivar1* es menor o igual que 5, saldrá un texto en pantalla, en una coordenada concreta que dirá "Menor o igual que 5". En cambio, si *mivar1* es mayor que 5, saldrá otro texto, en las mismas coordenadas, que diga "Mayor que 5".

## *Funcionamiento global del programa

El programa básicamente, pues, hace lo siguiente. Nada más comenzar se introduce en un bucle infinito (LOOP), por lo que, ya que no vemos ninguna sentencia BREAK, sabemos que este programa en principio nunca finalizará su ejecución – a no ser que nosotros le demos al botón de cerrar de la ventana (unas cuantas veces, eso sí...y de una forma no muy elegante, pero ya solucionaremos ese detalle más adelante). Nada más entrar en el bucle,

borramos todo el texto que pueda haber. Recordad que ponemos primero el **delete_text()** antes de cualquier **write()** porque si lo hiciéramos al revés, al llegar a la sentencia FRAME; no habría ningún texto que mostrar y no saldría nada en pantalla: hemos de recordar siempre de dejar justo justo antes del FRAME todo lo que queremos mostrar tal como lo queremos mostrar. Seguidamente ejecutamos la función **rand()**, que nos devuelve un entero aleatorio entre 1 y 10 y lo asignamos a una variable declarada previamente como privada. A partir de entonces esa variable tiene un valor que puede ser cualquiera entre 1 y 10. Seguidamente se realiza la comprobación mediante el IF/ELSE del valor de esa variable, y si es mayor o igual que 5 se escribirá un texto en pantalla y si es menor se escribirá otro, pero eso no se hará hasta que se llegue a FRAME. Una vez que salimos de ejecutar la correspondiente línea del bloque IF/ELSE, es cuando aparece precisamente FRAME, porque deseamos poner en pantalla en este momento el resultado del proceso del programa: es decir, la impresión de la frase de la sección IF/ELSE que se haya ejecutado. Llegado este punto, la iteración acaba y comenzamos una nueva iteración: borramos el texto, asignamos a la variable un nuevo valor aleatorio que no tendrá nada que ver con el que tenía antes, y se vuelve a hacer la comprobación. Como los valores que tendrá *mivar1* en las distintas iteraciones son aleatorios, unas veces serán mayores o iguales que 5 y otras veces serán menores que 5, por lo que en algunas iteraciones aparecerá una frase y otras la otra, mostrando así el efecto que vemos por pantalla: la frase cambia rápidamente de una en otra casi sin tiempo para leerla. Y así hasta el infinito.

¿Qué pasaría si quitáramos –comentáramos- el bloque LOOP/END? Pues que la ejecución del programa dura un suspiro, ya que sólo se ejecuta una sóla vez su código –no hay iteraciones que se repitan-, y por lo tanto, sólo se ejecuta una vez el FRAME, y por lo tanto, sólo se muestra en una fracción de segundo lo que sería una única frase, antes de llegar al END del programa.

## Otros bloques de control de flujo: Switch, for, from, repeat

En los dos programas anteriores hemos visto diferentes maneras de hacer bucles y condiciones. En concreto hemos visto los bucles WHILE/END y LOOP/END y el condicional IF/ELSE/END. Pero en Bennu hay más bloques que permiten la existencia de iteraciones o condiciones. En general, a todos estos bloques se les llama bloques de control de flujo, porque controlan dónde tiene que ejecutar el programa la siguiente instrucción, es decir, controlan el devenir del programa y dicen cuál es la siguiente línea a ejecutar.

Vamos a ver ahora los bloques de control de flujo que nos faltan para poder tener ya todas las herramientas necesarias para poder escribir programas flexibles.

### *Bloque "Switch/End"

```
SWITCH ( variable o expresión )
        CASE valores:
                Sentencias;
        END
        CASE valores:
                Sentencias;
        END
        CASE valores:
                Sentencias;
        END
        …
        DEFAULT:
                Sentencias;
        END
END
```

Una sentencia SWITCH/END consta en su interior de una serie de secciones CASE/END y, opcionalmente, una sección DEFAULT/END.

El SWITCH/END es como una especie de Mega-IF. Cuando se ejecuta una sentencia SWITCH, primero se

comprueba el valor de la variable o expresión que hay entre los paréntesis,y después, si el resultado está dentro del rango de valores contemplados en la primera sección CASE, se ejecutarán las sentencias de la misma y se dará por finalizada la sentencia. En caso de no estar el resultado de la expresión en el primer CASE se pasará a comprobarlo con el segundo CASE , el tercero, etc. Y por último, si existe una sección DEFAULT y el resultado de la expresión no ha coincidido con ninguna de las secciones CASE, entonces se ejecutarán las sentencias de la sección DEFAULT.

En una sección CASE se puede especificar:
-Un valor
-Un rango de valores **mínimo..máximo** (notar que estos valores extremos se separan por 2 puntos, no 3)
ó -Una lista de valores y/o rangos separados por comas.

Hay que hacer notar que una vez ejecutada una de las secciones CASE de una sentencia SWITCH ya no se ejecutarán más secciones CASE, aunque éstas especifiquen también el resultado de la expresión.

No es necesario ordenar las secciones CASE según sus valores (de menor a mayor, o de mayor a menor), pero sí es imprescindible que la sección DEFAULT (en caso de haberla) sea la última sección, y no puede haber más que una sección DEFAULT.

Es posible anidar sentencias SWITCH sin ningún límite, es decir, se pueden poner nuevas sentencias SWITCH dentro de una sección CASE (y cualquier otro tipo de sentencia).

Un ejemplo:

```
Import "mod_text";

Process Main()
private
        byte mivar1=220;
end
begin
        loop
            delete_text(0);
            switch (mivar1)
                case 0..3:
                  write(0,200,100,4,"La variable vale entre 0 y 3, incluidos");
                end
                case 4:
                  write(0,200,100,4,"La variable vale 4");
                end
                case 5,7,9:
                  write(0,200,100,4,"La variable vale 5 o 7 o 9");
                end
                default:
                  write(0,200,100,4,"La variable vale cualquier otro valor diferente de los anteriores");
                end
            end // switch
            frame;
        end //loop
end // begin
```

## *Bloque "For/End". Parámetro de la orden "Frame" . Diferencias entre "Write()" y "Write_var()"

```
FOR (valor_inicial_contador; condicion_final_bucle; incremento_contador)
                        Sentencias;
END
```

La sentencia FOR es una sentencia que implementa un bucle. Se deben especificar, entre paréntesis, tres

partes diferentes, separadas por símbolos ; (punto y coma) tras la palabra reservada FOR. Estas tres partes son opcionales (pueden omitirse) y son las siguientes:

-Valor inicial del contador:. En esta parte se suele codificar una sentencia de asignación que fija el valor inicial de la variable que va a utilizarse como contador de iteraciones del bucle. Un ejemplo puede ser la sentencia de asignación x=0; que fijaría la variable x a cero a inicio del bucle (valor para la primera iteración).

-Condición final del bucle:. En esta parte se especifica una condición; justo antes de cada iteración se comprobará que sea cierta para pasar a ejecutar el grupo de sentencias. Si la condición se evalúa como falsa, se finalizará el bucle FOR, continuando el programa tras el END del bucle FOR. Un ejemplo de condición puede ser x<10; que permitirá que se ejecute el grupo interior de sentencias únicamente cuando la variable x sea un número menor que 10.

-Incremento del contador: En la última de las tres partes es donde se indica el incremento de la variable usada como contador por cada iteración del bucle; normalmente, ésto se expresa también con una sentencia de asignación. Por ejemplo, la sentencia x=x+1; le sumaría 1 a la variable x tras cada iteración del bucle.

Tras la definición del bucle FOR, con sus tres partes, es donde debe aparecer el grupo de sentencias interiores del bucle que se van a repetir secuencialmente mientras se cumpla la condición de permanencia (parte segunda). Tras este grupo de sentencias la palabra reservada END determinará el final del bucle FOR.

Cuando en un programa llega a una sentencia FOR se ejecuta primero la parte de la inicialización y se comprueba la condición; si ésta es cierta se ejecutará el grupo de sentencias interiores y, después, la parte del incremento, volviéndose seguidamente a comprobar la condición, etc. Si antes de cualquier iteración la condición resulta falsa, finalizará la sentencia FOR inmediatamente.

Como se ha mencionado, las tres partes en la definición del bucle son opcionales; si se omitieran las tres, sería equivalente a un bucle LOOP...END.

En un bucle FOR pueden ponerse varias partes de inicialización, condición o incremento separadas por comas, ejecutándose todas las inicializaciones primero, luego comprobándose todas las condiciones de permanencia (si cualquiera resultara falsa, el bucle finalizaría), las sentencias interiores, y al final, tras cada iteración, todos los incrementos.

Una sentencia BREAK dentro de un bucle FOR lo finalizará de forma inmediata, continuando el programa por la sentencia siguiente a dicho bucle. Una sentencia CONTINUE dentro de un bucle FOR forzará al programa a ejecutar directamente la parte del incremento y, después, realizar la comprobación de permanencia y, si ésta es cierta, volver a ejecutar las sentencias interiores desde el principio. Si la condición resulta falsa, la sentencia CONTINUE finalizará el bucle FOR.

Las sentencias interiores a un bucle FOR pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles FOR.

Un ejemplo:

```
Import "mod_text";

Process Main()
private
        byte mivar1;
end
begin
      for (mivar1=1; mivar1<10; mivar1=mivar1+1)
            write(0,(mivar1*25)+30,100,4,mivar1);
            frame(2000);
      end
      loop
           frame;
      end
```

```
end
```

Este ejemplo tiene bastante miga. Vamos a repasarlo línea por línea. Nada más empezar el programa, nos metemos en el bucle FOR. Lo primero que ocurre allí es que asignamos el valor de 1 a *mivar1*, declarada previamente como de tipo BYTE. Después comprobamos si el valor de *mivar1* es menor que 10. Evidentemente, 1<10, y por lo tanto, se ejecutarán las sentencias internas del for (en la primera iteración no se tiene en cuenta para nada la sección del FOR tras el último ;). Una vez ejecutadas, se volverá arriba y antes de nada, se efectuará, ahora sí, el incremento, pasando así *mivar1* a valer 2. (Fijarse que a partir de la segunda iteración ya no se usa para nada la sección del FOR de delante del primer ;).Después de hacer el incremento, se volverá a comprobar la condición. Como todavía 2<10, se volverá a ejecutar el interior del FOR. Una vez hecho esto, se vuelve para arriba, se incrementa *mivar1* para que valga 3, se vuelve a comprobar la condición, y si es verdadera, se vuelve a ejecutar el interior del FOR, y así hasta que llegue una iteración donde al comprobar la condición ésta sea falsa. En ese momento, la ejecución del programa seguirá en la línea inmediatamente posterior al END del FOR.

Las sentencias interiores del FOR son dos. La primera lo que hace es escribir el valor de *mivar1*, que cambia en cada iteración, en posiciones diferentes según también el valor de *mivar1*, de manera que se vea claramente los incrementos y el momento donde el bucle acaba. Primero se imprime el número 1(el primer valor de *mivar1*), después, un poco más a la derecha –fijarse en el segundo parámetro del **write()**- se imprime el número 2 (el nuevo valor de *mivar1*), y así, hasta imprimir el número 9. Es importante recalcar que el número 10 no se imprimirá, porque, tal como hemos comentado, el proceso es primero incrementar, y luego hacer la comprobación: en la última iteración tendríamos que *mivar1* vale 9, entonces se va arriba, se incrementa a 10, se comprueba que 10 no es menor que 10, y se sale del FOR, por lo que cuando *mivar1* vale 10 ya no tiene la oportunidad de ser imprimida. Puedes jugar con los valores iniciales del contador, la condición de finalización o los valores de incremento. Así lo verás más claro.

La otra sentencia interior del FOR es la conocida FRAME para que cada vez que se realice una iteración, en la cual se quiere escribir un valor diferente de *mivar1*, se haga efectivo esa impresión. Como siempre, FRAME está dentro de un bucle, en este caso el FOR. Lo diferente es ese parámetro que no conocíamos que tuviera. ¿Para qué sirve? Primero prueba de quitar ese parámetro y dejar el FRAME a secas como siempre. Verás que el programa se ejecuta tan rápidamente que casi no hay tiempo de ver que los números van apareciendo uno seguido de otro. El parámetro del FRAME lo que indica es cuánto tiempo ha de pasar para que el intérprete de Bennu, una vez que ha llegado a la línea del FRAME, espere a que el "agujero" se abra y muestre el resultado del programa. Es decir, si no ponemos ningún parámetro, o ponemos el valor 100 –es lo mismo-, cada vez que se llegue a la línea FRAME, la "puerta" se abre y se muestra lo que hay inmediatamente. Este proceso es lo que se llama imprimir el siguiente fotograma. El fotograma es la únidad de tiempo utilizada en Bennu, (profundizaremos en ello en el siguiente capítulo), pero imagínate por ahora que el ritmo de aparición de los distintos fotogramas es como un reloj interno que tiene el programa, y que viene marcado por FRAME: cada vez que se "abre la puerta", se marca el ritmo de impresión por pantalla. De ahí el nombre al comando FRAME: cada vez que se llega a ese comando es como gritar "¡Mostrar fotograma!".Si se pone otro número como parámetro del FRAME, por ejemplo el 500, lo que ocurrirá es que cuando se llegue a la línea FRAME, la "puerta" no se abrirá inmediatamente sino que se esperará 5 fotogramas para abrirse. Es como si el programa se quedara esperando un rato más largo en la línea FRAME, hasta ejecutarla y continuar. Así pues, este parámetro permite, si se da un valor mayor de 100, ralentizar las impresiones por pantalla. En el caso concreto de nuestro ejemplo, pues,cada iteración tardará en mostrarse 20 veces más que lo normal (el parámetro vale 2000) porque el programa dejará pasar 20 fotogramas antes de mostrar por pantalla el resultado de dicha iteración. Si quieres, puedes cambiar su valor. También es posible poner un número menor de 100. En ese caso, lo que estaríamos haciendo es "abrir la puerta" más de una vez en cada fotograma: si pusiéramos por ejemplo 50, estaríamos doblando la velocidad normal de impresión por pantalla. Puedes comprobarlo también. De todas maneras, para entender mejor esto último que estamos comentando, en un apartado del cuarto capítulo de este manual he escrito un pequeño código que creo que te podrá ayudar a entender mejor el sentido de este parámetro que puede llevar el frame, aunque en este momento este programa contiene elementos que no hemos visto todavía.

Volvamos al código que teníamos. Fíjate que después de haber escrito el bucle FOR, el programa no acaba ahí sino que se añade un bucle LOOP con una única sentencia FRAME. Prueba de quitar ese LOOP. Verás que el programa se ejecuta pero cuando ya se ha impreso todos los valores de *mivar1*, evidentemente, el programa se acaba. Si queremos mantener el programa permaneciendo visible aún cuando se haya acabado el bucle del FOR, nos tenemos que inventar un sistema que también sea un bucle, infinito, cuya única función sea seguir mostrando el contenido de la pantalla tal cual. Esto es justo lo que hace el LOOP: va ejecutando el FRAME infinitamente para poder seguir visualizando la lista de números.

Por cierto, ¿qué pasaría si ahora comentamos la línea *frame(2000);*? Verás que el programa se ejecuta igual, pero que los números no van apareciendo por pantalla uno seguido de otro, sino que ya aparecen de golpe los 10

números. Esto es fácil de entender: si eliminamos la orden frame del bucle FOR, lo que estamos haciendo es evitar que en cada iteración se muestre por pantalla el resultado de dicha iteración justamente. Es decir, si eliminamos el frame, el bucle se realizará igual, iteración a iteración, pero hasta que no encuentre un frame no va a poder visualizar el resultado de sus cálculos. Por eso, cuando el programa entra en el Loop y se encuentra un frame (el primero), pone en pantalla de golpe todo aquello que llevaba haciendo hasta entonces, que es precisamente todas y cada una de las iteraciones del bucle FOR.

En apartados anteriores he comentado que para poder imprimir variables y no textos fijos se tenía que utilizar el comando **write_var()** en vez de **write()**, cosa que no he hecho en este ejemplo. ¿Por qué? En realidad, el comando **write()**, tal como acabamos de comprobar, sí que permite la impresión de valores de variables, incluso podríamos escribir algo así como

```
mivar1=10;
write(0,100,100,4,"Esta variable vale:" + mivar1);
```

donde puedes ver de paso que el signo + también sirve para poder unir trozos de frases y valores separados para que aparezcan como un único texto (acabamos de descubrir pues que el signo + es en Bennu el llamado técnicamente operador de "concatenación" de cadenas). La razón de haber elegido en este ejemplo la función **write()** en vez de **write_var()** viene de su principal diferencia de comportamiento, ya explicada anteriormente, respecto la impresión por pantalla de valores variables. Recordemos que con **write()** una vez que se escribe el valor de la variable, éste pasa a ser un valor fijo en pantalla como cualquier otro y no se puede cambiar y que en cambio, con **write_var()**, los valores de las variables que se hayan escrito, a cada fotograma se comprueba si han cambiado o no, y si lo han hecho, se actualiza automáticamente su valor. Es decir, que con **write_var()** los valores impresos de las variables cambiarán al momento que dichos valores cambien en el programa, permanentemente. Esto lo puedes comprobar: si cambiar la función **write()** por **write_var()**, verás que a cada iteración se va imprimiendo un nuevo número, pero los números impresos anteriormente también cambian al nuevo valor: esto es así porque como estamos cambiando los valores a *mivar1*, todos los valores de esa variable, impresos en cualquier momento de la ejecución del programa cambiarán también a cada fotograma que pase.

Un detalle curioso es ver que, si usamos **write_var()**, sí que se llega a imprimir el número 10. ¿Por qué? Porque, tal como hemos comentado, en la última iteración *mivar1* vale 9, entonces se sube arriba, se le incrementa a 10, se comprueba que sea cierta la condición, y al no serlo se sale del for. Entonces, aparentemente el 10 no se tendría que escribir, ¿verdad? La clave está en darse cuenta que *mivar1* llega a valer 10 en el momento de salir del FOR. Y como seguidamente nos encontramos con un LOOP, este bucle nos permite continuar con la ejecución del programa, y por tanto, permite la aparición de sucesivos fotogramas. Como he dicho que a cada fotograma del programa se realiza una comprobación y actualización automática de todos los valores impresos por **write_var()**, como en ese momento *mivar1* vale 10, cuando se llega la primera vez al FRAME del LOOP, se llega a un fotograma nuevo, y así, se realiza la actualización masiva de todos los números. A partir de allí, como el programa ya únicamente consiste en ir pasando fotogramas y ya está, el valor 10 de *mivar1* va a continuar siendo el mismo y ya no se ve ningún cambio en pantalla.

Una última apreciación: fíjate que el bucle FOR no es imprescindible para programar. Cualquier bucle FOR se puede sustituir por un bucle WHILE. Fíjate en los cambios que se han producido al ejemplo que estamos trabajando: hemos cambiado el FOR por un WHILE, pero el programa sigue siendo exactamente el mismo. Es fácil de ver:

```
Import "mod_text";

Process Main()
private
        byte mivar1;
end
begin
      mivar1=1;
      while (mivar1<10)
            write(0,(mivar1*25)+30,100,4,mivar1);
            mivar1=mivar1+1;
            frame(2000);
      end
      loop
           frame;
```

```
      end
end
```

Y ya para acabar, a ver si sabes qué hace este código antes de ejecutarlo:

```
Import "mod_text";

Process Main()
private
          int a;
         int b;
end
begin
      for (a=0,b=10;a<5 AND b>5;a=a+1,b=b-1)
            write(0,0,10*a,0,"a="+a+" b="+b) ;
      end
      loop
            frame;
       end
end
```

## *Bloque "Repeat/Until"

**REPEAT**
**Sentencias;**
**UNTIL ( condición )**

La sentencia REPEAT/UNTIL es una sentencia muy similar a WHILE. Debe comenzar con la palabra reservada REPEAT, seguida de las sentencias que se quieren repetir una o más veces y el final de la sentencia se determinará poniendo la palabra reservada UNTIL –"hasta"- seguida de la condición que se debe cumplir para que se dé por finalizada la sentencia.

Cuando se ejecute una sentencia REPEAT se ejecutarán primero las sentencias interiores (las que están entre el REPEAT y el UNTIL) y, tras hacerlo, se comprobará la condición especificada en el UNTIL y si ésta continúa siendo falsa, se volverán a ejecutar las sentencias interiores. El proceso se repetirá hasta que la condición del UNTIL resulte cierta, continuando entonces la ejecución del programa en la sentencia siguiente al bucle.

Las sentencias interiores a un bucle REPEAT pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles REPEAT.

Una sentencia BREAK dentro de un bucle REPEAT lo finalizará de forma inmediata, continuando el programa a partir de la sentencia siguiente a dicho bucle. Una sentencia CONTINUE dentro de un bucle REPEAT forzará al programa a comprobar la condición del UNTIL inmediatamente, y si ésta es falsa, volverá a ejecutar las sentencias interiores desde el principio (tras la palabra reservada REPEAT). Si la condición resulta cierta, la sentencia CONTINUE finalizará el bucle.

Cada vez que se ejecutan las sentencias interiores se dice que se ha realizado una iteración del bucle. La sentencia REPEAT...UNTIL (literalmente traducida como REPETIR...HASTA (que se cumpla la) condición) siempre ejecutará las sentencias interiores al menos una vez, ya que comprueba la condición siempre tras ejecutarlas. Ahí está de hecho la diferencia con el bucle WHILE. Como en el WHILE la condición se comprueba antes de ejecutar las sentencias interiores, puede darse el caso que la primera vez que se llegue a un WHILE la condición ya sea directamente falsa, y por tanto, las sentencias interiores no se lleguen a ejecutar nunca. En cambio, como en el REPEAT la condición se comprueba después de ejecutar las sentencias interiores, como mínimo éstas siempre se ejecutarán una vez, porque no se comprobará la falsedad de la condición hasta después se haber ejecutado dichas sentencias interiores. Esta diferencia entre un WHILE/END y un REPEAT/UNTIL la puedes ver en estos ejemplos:

```
Import "mod_text";
```

```
Process Main()
private
      byte mivar1=1;
end
begin
      while(mivar1<1)
            write(0,100,100,4,mivar1);
            frame;
      end
      loop
           frame;
      end
end
```

```
Import "mod_text";

Process Main()
private
      byte mivar1=1;
end
begin
      repeat
           write(0,100,100,4,mivar1);
           frame;
      until (mivar1<1)
      loop
           frame;
      end
end
```

En el primer ejemplo sólo veremos una pantalla negra, y en el segundo el número 1 impreso. Esto es porque en el primer ejemplo, antes de nada, se comprueba si la condición es verdadera. Como no lo es, el while/end no se ejecuta y se pasa directamente al Loop, donde no hay nada que mostrar. En cambio, en el segundo ejemplo, primero se realiza el **write()** y el frame siguientes al repeat, y después se comprueba la condición. Al ser falsa, no se realiza ninguna iteración más y se va a loop, pero como mínimo se ha impreso el número debido a la única ejecución del **write()** que ha existido.

## *Bloque "From/End"

```
FROM variable = valor_num TO valor_num STEP valor_num ;
            Sentencias;
END
```

La sentencia FROM implementa un bucle también. Para ello, se necesita una variable que sirva como contador del bucle. De hecho, su funcionamiento es casi idéntico al FOR/END, lo único que el FROM tiene una sintaxis más sencilla y sólo sirve para bucles incrementales/decrementales mientras que el FOR permite muchísimos otros usos gracias a que se puede definir la comparación y la función de incremento.

Antes de las sentencias que conformarán el grupo interior de sentencias se debe poner la palabra reservada FROM seguida del nombre de la variable contador, el símbolo de asignación (=), el valor inicial de la variable, la palabra reservada TO y, finalmente, el valor final de la variable. Tras esta declaración del bucle FROM, extrañamente, se debe poner el símbolo ; (punto y coma). Después de esta cabecera definiendo las condiciones del bucle vendrá el grupo interior de sentencias que se pretende repetir un número determinado de veces y, al final, la palabra reservada END.

La primera iteración se hará con el valor inicial en la variable usada como contador; tras esta iteración se

le sumará 1 a esta variable (si el valor inicial es menor que el valor final) o se le restará 1 (en caso contrario). Tras actualizar el valor de la variable, se pasará a la siguiente iteración siempre que el valor de la variable no haya llegado (o sobrepasado) el valor final del bucle.

Las sentencias interiores a un bucle FROM pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles FROM.

Una sentencia BREAK dentro de un bucle FROM lo finalizará de forma inmediata, continuando el programa por la sentencia siguiente a dicho bucle (tras el END).Una sentencia CONTINUE dentro de un bucle FROM forzará al programa a incrementar inmediatamente la variable usada como contador y, después, si no se ha sobrepasado el valor final, comenzar con la siguiente iteración.

Como opción, es posible poner tras los valores inicial y final de la sentencia, la palabra reservada STEP seguida de un valor constante que indique el incremento de la variable contador tras cada iteración del bucle, en lugar de +1 o -1, que son los incrementos que se harán por defecto si se omite la declaración STEP (paso).

Los valores inicial y final de un bucle FROM deben ser diferentes. Si el valor inicial es menor que el valor final, no se puede especificar un valor negativo en la declaración STEP, y viceversa.

## Profundizando en la escritura de texto en la pantalla: ”Move_text()”,”Text_height()”, “Text_width()” y TEXT_Z

Hasta ahora para trabajar con textos en pantalla hemos utilizado las funciones **write()** ,**write_var()**, **delete_text()** y poco más. Pero hay más funciones relacionadas con los textos.

Por ejemplo, tenemos **move_text()** ,que sirve para mover a otras coordenadas de la pantalla un texto ya impreso antes, cuyo identificador (generado por el **write()** utilizado la primera vez que se imprimió ese texto) se ha de pasar como parámetro (además de las coordenadas adonde se quiere mover,claro)

Recordemos el primer código que aprendimos en este curso:

```
Import “mod_text”;

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end
end
```

Recuerda que lo que hacía este programa era hacer un scroll horizontal de la frase “¡Hola mundo!”. Podríamos ahora modificar este programa para que haga exactamente lo mismo, pero utilizando la función **move_text()**. Es más, esta nueva versión incluso sería “mejor” que la primera porque se pueden evitar los posibles parpadeos que se producen cuando se borra y escriben los textos constantemente a una elevada velocidad, tal como se hace en el programa original. Bien, pues la versión con **move_text()** será así:

```
Import “mod_text”;

Process Main()
Private
          int mivar1;
End
```

```
Begin
          mivar1=10;
          write(0,10,100,1,"¡Hola mundo!");
          while(mivar1<320)
                    mivar1=mivar1+2;
                    move_text(1,mivar1,100);
                    frame;
          end
end
```

**Move_text()** tiene tres parámetros: el identificador del texto que se quiere mover, y la nueva coordenada X y Y donde se quiere mover. Fíjate que como valor del primer parámetro en el ejemplo ponemos un 1, porque recuerda que **write()** devuelve siempre (aunque nosotros no lo recojamos) el identificador único del texto que escribe, y siempre empieza por el número 1 y va aumentando (2,3…) por cada nuevo texto que se imprime. Así pues, en vez de borrar y reescribir el mismo texto, simplemente se escribe una sóla vez y se mueve a la coordenada (*mivar1*,100).

Fíjate además en otro detalle. Para poder utilizar **move_text()** hemos tenido que escribir previamente el texto con **write()** (para conseguir su identificador). Es decir, que el funcionamiento es imprimir una vez el texto en una posición inicial de pantalla, y posteriormente, irlo moviendo. Por tanto, el **write()** lo hemos sacado fuera del while, porque sólo imprimiremos el texto una sóla vez, la primera.Y el **delete_text()** lo hemos quitado porque ya no hace falta.

Modifiquemos el ejemplo anterior añadiendo un segundo texto en pantalla.Fíjate que el nuevo texto permanecerá fijo porque el **move_text()** sólo afecta al texto con código 1, y como "¡Y adiós!" se imprime después de "¡Hola mundo!", "¡Y adiós!" tendrá automáticamente el código 2.:

```
Import "mod_text";

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          write(0,10,100,1,"¡Hola mundo!");
          write(0,10,150,1,"¡Y adiós!");
          while(mivar1<320)
                    mivar1=mivar1+2;
                    move_text(1,mivar1,100);
                    frame;
          end
end
```

Por último, un ejercicio. ¿Cómo haríamos para hacer que el texto "¡Y adiós!", que ahora está fijo, realice un scroll horizontal, y que lo además haga de derecha a izquierda? Pues así:

```
Import "mod_text";

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          write(0,10,100,1,"¡Hola mundo!");
          write(0,250,50,1,"¡Y adiós!");
          while(mivar1<320)
                    mivar1=mivar1+2;
                    move_text(1,mivar1,100);
                    move_text(2,320-mivar1,50);
                    frame;
          end
end
```

Hacemos que el segundo texto (código identificador 2) se mueva también,procurando que la coordenada X inicial sea la más a la derecha posible (320) y que en cada iteración vaya disminuyendo para así irse moviendo hacia la izquierda. Cuando *mivar1* valga 0, el primer texto se imprimirá en (0,100) y el segundo en (320,50); cuando *mivar1* valga 100 el primer texto se imprimirá en (100,100) y el segundo en (220,50), cuando valga 320 el primer texto se imprimirá en (320,100) y el segundo en (0,50)…

Otro ejemplo un poco más avanzado y bonito: un texto oscilante. En este caso utilizamos junto con **move_text()** la función **get_disty()** -incluida en el módulo "mod_math"-, que no me hemos visto y que se estudiará más adelante.

```
Import "mod_text";
Import "mod_math";

Process Main()
Private
          int var=10;
          int idtexto;
End
Begin
    idtexto=write(0,100,30,4,"texto oscilante");
    Loop
        var=var+22500;
        move_text(idtexto,100+get_disty(var,20),30);
        Frame;
    End
end
```

También disponemos en el módulo "mod_text" de las funciones **text_height()** y **text_width()**, dos funciones que, pasándoles como primer parámetro el identificador correspondiente a la fuente de letra que se quiera utilizar (si no queremos usar ninguna concreta y con la predefinida del sistema nos basta, tendremos que escribir un 0), y como segundo parámetro un texto cualquiera, nos devuelven el alto y el ancho, respectivamente, de ese texto en píxeles (es decir, la altura o ancho del carácter más grande que vaya a dibujarse de toda la cadena, teniendo en cuenta la fuente elegida), pero sin llegar a escribir nada: algo importante para calcular en qué posición escribir ese u otro texto, antes de hacerlo de forma efectiva. Un ejemplo de uso:

```
Import "mod_text";

Process Main()
Private
      int anchura;
      int altura;
End
Begin
        anchura=text_width(0,"Esto es un texto");
        altura=text_height(0,"Esto es un texto");
        write(0,100,50,4,anchura);
        write(0,100,100,4,altura);
        Loop
              Frame;
        End
end
```

Por último, comentar que la existencia no ya de una orden, sino de una variable entera (global predefinida): **TEXT_Z** . Al modificar su valor, estaremos cambiando la profundidad a la que se escribirán los textos a partir de entonces. Esta variable es útil por si nos interesa que los textos queden detrás o delante de algún gráfico (ya que éstos, ya lo veremos, también tienen una profundidad regulable), o detrás del fondo para hacerlos invisibles, por ejemplo. Cuanto más "atrás", más alejado queramos poner el texto, mayor será tendrá que ser su valor -hasta el máximo que permite el tipo Int- , y cuanto más "adelante", más cercano queramos escribirlo, menor tendrá que ser. Por defecto, si no se indica nada, los textos se escriben con un valor de TEXT_Z igual a -256.

Que esta variable sea "predefinida" significa que no es necesario declararla en ningún sitio: se puede usar directamente con ese nombre específico en cualquier parte de nuestro código. Eso sí, para la tarea que tiene asignada

(cambiar la profundidad del texto), no para otra cosa. Lo de que sea "global" no nos importa ahora (ya se explicará en posteriores capítulos) simplemente hay que saber que la puedes usar con ese nombre cuando quieras y donde quieras para realizar, eso sí, una tarea muy concreta.

## Síntesis del capítulo: algunos ejemplos más de códigos fuente

A continuación mostraremos unos cuantos códigos fuente, sin explicarlos, a modo de síntesis de todo lo que ha explicado hasta ahora. En estos códigos no se introduce ningún concepto nuevo: han de servir como refresco y ejemplo práctico de todo lo leído hasta ahora. Si se han comprendido las ideas básicas de programación (condicionales, bucles,variables…) no debería de ser difícil adivinar qué es lo que hacen cada uno de los códigos siguientes, (antes de ejecutarlos, por supuesto).

**Primer ejemplo:**

```
Import "mod_text";
Import "mod_rand";

Process Main()
private
          int mivarX=33;//Un valor inicial cualquiera
          int mivarY=174; //Un valor inicial cualquiera
end
begin
          loop
//Si no se especifica nada,las dimensiones por defecto de la pantalla son siempre 320x200
                    mivarX=rand(0,320);
                    mivarY=rand(0,200);
                    if(mivarX>100 and mivarX <250 and mivarY>50 and mivarY<150)
                              continue;
                    end
                    write(0,mivarX,mivarY,4,".");
                    frame;
          end
end
```

Con este código se podría jugar a poner un **delete_text(0)** justo antes del **write()**,para ver qué efecto curioso da (y así tampoco dará el error comentado en apartados anteriores de tener demasiados textos en pantalla, con la consecuencia de congelar por completo la ejecución de nuestro programa y "colgarlo"), aunque haciendo esto no se notará el efecto del bloque IF/END.

**Segundo ejemplo:**

```
Import "mod_text";

Process Main()
private
          int columna;
          int fila;
end
begin
          for(columna=1;columna<=5;columna=columna+1)
                    for(fila=0;fila<5;fila=fila+1)
                              write(0,(columna*20)+30,(fila*20)+30,4,"*");
                              frame;
                    end
          end
          loop
                    frame;
          end
end
```

Con este código también se podría jugar a poner un **delete_text(0)** delante del **write()** para observar mejor cómo cambian en las distintas iteraciones los valores de las variables *columna* y *fila*.

Explico un poco el código: fíjate que hay dos FOR anidados: esto implica que por cada iteración que se produzca del FOR externo *for(columna=1;...)* -en total son 5: de 1 a 5-, se realizarán 5 iteraciones del FOR interno *for(fila=1;...)* –de 0 a 4-, por lo que se acabarán haciendo finalmente 25 iteraciones con los valores siguientes: columna=1 y fila=0, columna=1 y fila=1, columna=1 y fila=2, columna=1 y fila=3, columna=1 y fila=4, columna=2 y fila=0, columna=2 y fila=1, etc. Luego ya sólo basta imprimir el signo * en las coordenadas adecuadas para dar la apariencia de cuadrado.

Un detalle interesante es observar qué pasa si eliminamos la línea **frame;** del interior de los bucles FOR. El cuadrado se verá igual, pero de golpe: no se verá el proceso de impresión gradual de los asteriscos. Esto es fácil de entender. Si mantenemos la orden **frame** dentro de los FOR, en cada iteración estaremos obligando al programa a mostrar por pantalla el resultado de los cálculos que ha realizado desde el último frame anterior. Pero en cambio, si no lo ponemos, los cálculos se realizarán igualmente –las iteraciones y la orden **write()** se ejecutarán- pero no se mostrará ningún resultado visible hasta que se encuentre con alguna orden **frame;**,orden que aparece en el bloque LOOP/END del final (necesario para permitir que el programa continúe ejecutándose indefinidamente y no se acabe ipsofacto). Por tanto, cuando, una vez que el programa haya salido de los FOR anidados, se encuentre la primera vez con el **frame;** del interior del LOOP, "vomitará" de golpe en la pantalla todo aquello que ha venido calculando y "escribiendo" hasta entonces, por lo que el cuadrado de asteriscos aparecerá directamente completo.

Una variante interesante de este código es el siguiente:

```
Import "mod_text";

Process Main()
private
          int columna;
          int fila;
end
begin
          for(columna=1;columna<=5;columna=columna+1)
                    for(fila=1;fila<=columna;fila=fila+1)
                              write(0,(columna*20)+30,(fila*20)+30,4,"*");
                              frame;
                    end
          end
          loop
                    frame;
          end
end
```

Lo único que se ha cambiado son los valores de los elementos que componen el FOR más interno. Fíjate que ahora este FOR realizará tantas iteraciones como el valor de "columna" le permita, porque ahora el límite máximo de iteraciones de este FOR viene impuesto por dicho valor. Es decir, que tendremos este orden de valores para las dos variables: columna=1 y fila=1, columna=2 y fila=1, columna=2 y fila=2, columna=3 y fila=1, columna=3 y fila=2,columna=3 y fila=3, etc. Por eso sale ese triángulo.

Si queremos que salga otro tipo de triángulo podríamos hacer:

```
Import "mod_text";

Process Main()
private
          int columna;
          int fila;
end
begin
          for(fila=1;fila<=5;fila=fila+1)
                    for(columna=1;columna<=fila;columna=columna+1)
                              write(0,(columna*20)+30,(fila*20)+30,4,"*");
                              frame;
                    end
          end
```

```
        loop
                frame;
        end
end
```

O bien,otra manera sería:

```
Import "mod_text";

Process Main()
private
        int columna;
        int fila;
end
begin
        for(fila=5;fila>=1;fila=fila-1)
                for(columna=5;columna>=fila;columna=columna-1)
                        write(0,(columna*20)+30,(fila*20)+30,4,"*");
                        frame;
                end
        end
        loop
                frame;
        end
end
```

Podemos "pintar" más figuras curiosas como ésta:

```
Import "mod_text";

Process Main()
private
        int i;
end
begin
        for(i=100;i<=200;i=i+10)
                        write(0,i,50,4,"*");
                        write(0,i,100,4,"*");
        end
        for(i=50;i<=100;i=i+10)
                        write(0,100,i,4,"*");
                        write(0,200,i,4,"*");
        end

        loop
                frame;
        end
end
```

O ésta:

```
Import "mod_text";

Process Main()
private
        int i;
end
begin
        for(i=75;i<=125;i=i+10)
                        write(0,i,65,4,"*");
        end
        for(i=50;i<=100;i=i+10)
                        write(0,100,i,4,"*");
        end

        loop
                frame;
```

```
        end
end
```

Y ya para acabar, un juego. Evidentemente, el siguiente código no escribirá nada en pantalla. ¿Por qué?

```
Import "mod_text";

Process Main()
private
        int columna;
        int fila;
end
begin
        for(fila=0;fila>=1;fila=fila-1)
                for(columna=5;columna>=fila;columna=columna-1)
                        write(0,(columna*20)+30,(fila*20)+30,4,"*");
                        frame;
                end
        end
        loop
                frame;
        end
end
```

**Tercer ejemplo:**

El siguiente ejemplo imprimirá por pantalla los 20 primeros números de la serie de Fibonacci.

La serie de Fibonacci es una secuencia de números enteros que tiene muchas propiedades matemáticas muy interesantes, y que en la Naturaleza se ve reflejada en multitud de fenómenos. Comienza con dos números 1. A partir de ahí, los números que forman parte de esta serie han de cumplir lo siguiente: ser el resultado de la suma de los dos números anteriores. Es decir, el siguiente número ha de ser la suma de los dos 1. Por tanto, 2. El siguiente número ha de ser la suma de un 1 con ese 2. Por tanto, 3. Y así. Es fácil ver, por tanto, que el resultado que queremos que obtenga nuestro programa ha de ser el siguiente:

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Antes de nada, hay que decidir qué bucle utilizaremos para imprimir estos números (porque está claro que utilizaremos un bucle, ¿verdad?). Como queremos exactamente 20 números, lo más sencillo y directo es utilizar un bucle FOR, porque con este bucle tenemos controlado cuántas iteraciones exactamente queremos realizar (en este caso, 20). Así que el código fuente es el siguiente:

```
Import "mod_text";

Process Main()
private
        int num=1;
        int num2=1;
        int suma;
        int cont;
end
begin
        //Imprimimos los dos primeros números de la serie (1 y 1)
        write(0,200,10,4,num);
        write(0,200,20,4,num2);
        for(cont=3;cont<=20;cont=cont+1)
                suma=num+num2;
                write(0,200,cont*10,4,suma);
/*Corremos los números."num" se pierde,"num2" pasa a ser "num" y "suma" pasa a ser "num2"*/
                num=num2;
                num2=suma;
                frame;
        end
        loop
```

```
                    frame;
          end
end
```

Otro ejemplo de programa "matemático", el cual escribe por pantalla los quince primeros números que cumplen ser múltiples de 3 pero no de 6:

```
Import "mod_text";

Process Main()
private
          int i=0;
          int cont=0;
end
begin
          loop
                    i=i+1;
                    if(i%3==0 and i%6!=0)
                                        write(0,20+3*i,100,4,i + ",");
                                        cont=cont+1;
                    end
                    if(cont==15) break; end
          end
          loop
                    frame;
          end
end
```

¿Sabrías decir antes de ejecutar este código, qué es lo que saldrá por pantalla?

```
Import "mod_text";

Process Main()
private
          int i=0;
          int result;
end
begin
          for(i=1;i<=20;i++)
                    result=i*i;
                    write(0,150,10*i,4,result);
          end
          loop
                    frame;
          end
end
```

¿Entiendes las operaciones matemáticas que se utilizan en este ejemplo?

```
Import "mod_text";

Process Main()
private
   int a=57;
end
begin
  write(0,160,90,4,"Tengo billetes de 5 euros y monedas de 2 y 1 euro.");
  write(0,160,100,4,"Para conseguir " + a + " euros necesito");
  write(0,160,110,4,a/5 + " billetes y " + (a%5)/2 + " monedas de 2 y " + (a%5)%2 + "
monedas de 1");
  loop
    frame;
  end
end
```

# Guía de Estilo del programador Bennu

**(documento propuesto por SplinterGU, Noviembre 2008 Rev. 3)**

Una “Guía de Estilo de Codificación” es un documento que pretende definir una manera estándar de escribir el código fuente mediante un determinado lenguaje de programación. La mayoría de lenguajes tienen escrita una, y Bennu también. El objetivo de esta guía es ayudar a construir programas correctos, entendibles y fáciles de mantener, (aunque trata exclusivamente del estilo de programación en cuanto a codificación se refiere: no se tratan cuestiones de diseño, organización funcional ni otras buenas prácticas en la construcción de programas). A continuación se expondrá una síntesis resumida de sus apartados más relevantes.

Para comenzar, la guía de estilo recomienda las siguientes extensiones para los nombres de archivos:

- código fuente en BGD: *.prg
- archivos de encabezados: *.h *.inc
- archivos de "make": Makefile

Respecto el tema de la documentación interna de los códigos fuente, se dice que todos los archivos de código fuente deberán tener al inicio del mismo una leyenda de licencia de los mismos, como así también el correspondiente copyright (autores, fecha, versión, etc.).Este texto debe tener el mismo formato en todo el proyecto, y se sugiere el siguiente formato:

```
/*
 *  Copyright - <Año/s>, <Autor>
 *
 *  This file is part of <Nombre del proyecto>
 *
 *  <Nombre del proyecto> is free software;
 *  you can redistribute it and/or modify it under
 *  the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  <Nombre del proyecto> is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 *
 */
```

*Año/s*, puede ser un solo año o un rango (separado por “-”) o una lista (separado por ”,”).Se sugiere también que de existir varios autores se divida la linea de “Copyright” en varias, una por autor:

```
/*
 *  Copyright - <Año/s>, <Autor 1>
 *  Copyright - <Año/s>, <Autor 2>
 *
 *  This file is part of <Nombre del proyecto>
 *
 *  <Nombre del proyecto> is free software;
 *  you can redistribute it and/or modify it under
 *  the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  <Nombre del proyecto> is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 *
 */
```

Además, cada función o rutina, deberá incorporar antes de la palabra clave PROCESS/FUNCTION una breve descripción de la misma, como así también descripción de cada uno de los parámetros de entrada, salida y retorno (si corresponde), y tipo de dato de cada uno de ellos. Por otra parte, si el nombre de los procesos/funciones es una combinación de palabras, se debe usar mayúsculas para la primer letra de cada palabra y un guión bajo para separar estas entre sí. Y no se debe usar nunca en los nombres de procesos o funciones caracteres acentuados ni especiales. Finalmente, se recomienda prefijar a los procesos/funciones con 3 o más letras en mayúsculas y un guión bajo, que identifiquen el grupo al cual pertenecen.

Respecto el tema de las variables, las declaradas en las secciones PUBLIC, LOCAL, PRIVATE, ó GLOBAL deben tener un comentario descriptivo de las mismas.Hay que evitar usar valores por defecto en la declaración, y hay que especificar siempre el tipo de dato de cada variable. También se deben prefijar las variables acorde a las siguientes reglas y en el orden en que se describen a continuación:

*Según su ámbito:

| Prefijo | Tipo de dato |
|---|---|
| g_ | GLOBAL |
| l_ | LOCAL |
| p_ | PUBLICA |
| _ | PRIVADA |

*Según su tipo:

| Prefijo | Tipo de dato |
|---|---|
| i | INT |
| d | DWORD |
| w | WORD |
| b | BYTE |
| c | CHAR |
| f | FLOAT |
| p | POINTER (sin importar el tipo de puntero) |
| s | STRING |

Después del prefijo de tipo de dato, se recomienda prefijar las variables globales con 3 o más letras en mayúsculas y un guión bajo, que identifiquen el fuente/grupo al cual pertenecen.Si el nombre de las variables es una combinación de palabras, se deben usar mayúsculas para la primer letra de cada palabra y un guión bajo para separar estas entre sí. Y no se debe usar nunca en los nombres de las variables caracteres acentuados ni especiales. Por ejemplo:

```
BYTE    g_bCONFIG_Inicializado ;
INT     p_iTotal_Eliminados;
CHAR    g_cKEY_Tecla_Arriba;
```

Finalmente, se deben declarar una variable por linea de código (es decir, evitar declarar múltiples variables por linea).

Respecto el tema de las constantes (CONST), éstas deben tener un comentario descriptivo de las mismas. Todas las palabras que conformen el nombre de una contante deben ser escritas en forma capitalizada (primera letra en mayúsculas, resto en minúsculas) y separadas entre sí por un guión bajo. No usar caracteres acentuados ni especiales en los nombres de las constantes. No es necesario prefijar las constantes, aunque de ser conveniente, puede hacerse.

Por otro lado, se recomienda no escribir más de 1 instrucción por linea de código. Es decir, no escribir por ejemplo:

```
if(x>10) x=11; y=12; end
```

sino

```
if(x>10)
    x=11;
    y=12;
end
```

Respecto el tema de las declaraciones d bloques, cada bloque de declaración (LOCAL, GLOBAL, PRIVATE, PUBLIC, DECLARE) debe ser cerrado con un END.

A continuación, se listan una serie de consejos generales sobre el uso de los espacios y líneas en blanco:

-Usar espacios en blanco a discreción para mayor claridad en el código.
-Al usar operadores deben usarse al menos un espacio de separación entre los mismos.
-Usar un espacio de separación luego de abrir un paréntesis y antes de cerrar éste.
-No usar espacios entre una función/proceso y el paréntesis de los parámetros.
-Usar un espacio de separación entre KEYWORD que usan paréntesis y estos. Por ejemplo: “if ( ... )”.
-Tras cada coma en una lista de argumentos o valores, debe usarse un espacio (o más si esto ayuda a una visualización más clara del código).
-Entre función o rutina, y antes de la descripción de ésta, se deben dejar 2 líneas en blanco.
-No dejar más de una linea de separación entre lineas diferentes de código, a menos que se quiera resaltar un bloque específicamente, caso en el que se deben usar solo 2 líneas en blanco (no excederse).
-No dejar líneas de separación tras la apertura de un bloque (PROCESS,FUNCTION,BEGIN,FOR,etc..)
-Dejar una línea de separación entre la última instrucción de un bloque y el correspondiente cierre de éste (END, UNTIL, etc)
-Evitar dejar espacios en blanco tras las líneas (esto se puede especificar por configuración en la mayoría de los editores).

Respecto las identaciones:

-Para las tabulaciones usar espacios en vez de TAB, estas tabulaciones deben ser de 4 (cuatro) espacios.
-Todas las instrucciones deben indentarse salvo las de preprocesamiento, las cuales deben estar alineadas a inicio de la linea.

Otras cosas a tener en cuenta:

-Cada fuente/encabezado debe comenzar (tras el copyright) con una breve descripción del mismo, en el

caso del fuente principal debe incluir una descripción del proyecto.

-No se deben usar paths absolutos en ningún caso que refiera a recursos del proyecto.

-Se deben usar mayúsculas para los nombres de los defines, y si éstos están formados por una combinación de palabras, se deben separar a éstas con un guión bajo.

-En el caso de usar valores constantes flotantes y estos no tener decimales, se les debe adicionar a éstos ".0", por ejemplo, en vez de usar "123" para un flotante de valor constante, se debe usar "123.0".

Se recomienda usar la cláusula **on_exit** en una función/proceso, para liberar aquellos recursos reservados por ésta.

El orden de las diferentes secciones (no es obligatoria la existencia de todas ellas) que puede tener nuestro código fuente (algunas de las cuales todavía no hemos estudiado) debe ser el siguiente:

a) IMPORT
b) DEFINE
c) CONST
d) TYPE
e) INCLUDE (tipos de datos externos si son necesarios en los siguientes items)
f) GLOBAL
g) LOCAL
h) DECLARE (de funciones/procesos locales usados en módulos externos)
i) INCLUDE (funciones/procesos externos)
j) DECLARE (locales y privadas al fuente)

# CAPITULO 3

## Gráficos

Todos los ejemplos que hemos hecho hasta ahora no han incluido ningún tipo de gráfico: sólo textos. Es evidente que para diseñar un juego necesitarás gráficos (además de posiblemente música y otros efectos). En este capítulo se tratará las diferentes maneras de incluir los gráficos de nuestro juego, y trabajar con ellos.

### *Los modos de pantalla. Orden "Set_mode()"

Hasta ahora, la pantalla donde se visualizaba cualquier programa era de un tamaño de 320x240 píxeles (y 32 bits de profundidad), ya que no habíamos indicado nada que dijera lo contrario, y ése es el tamaño y profundidad por defecto. Una de las cosas que debemos aprender primero precisamente será configurar el modo de pantalla, es decir, la resolución de nuestro juego. Esto es un paso importante porque determina muchas cosas del juego, como el tamaño de los gráficos o los recursos necesarios, pues a grandes resoluciones tenemos gran cantidad de pixeles (puntos) en la pantalla y gracias a eso conseguimos una mayor nitidez en la imagen y se reduce el efecto de bordes dentados o los escalones, pero a cambio, los gráficos deben ser más grandes y por lo tanto aumenta la cantidad de cálculos que tiene que hacer el ordenador.

Para establecer el tamaño y la resolución de nuestro juego (o sea, establecer el "modo gráfico"), se utiliza la función **set_mode()**, la cual está definida dentro del módulo "mod_video".

Es válido llamar a esta función aunque ya haya un modo gráfico en funcionamiento, sea para cambiar la resolución de pantalla, o pasar de ventana a pantalla completa, por ejemplo, aunque se recomienda llamar a **set_mode()** nada más comenzar el programa, para evitar efectos de parpadeo al iniciar el modo gráfico dos veces.

Esta función tiene cuatro parámetros, (los dos últimos opcionales). Los dos primeros son justamente la resolución horizontal y vertical del juego respectivamente. Algunos valores estándar (los cuales guardan las proporciones de pantalla normalizadas 4:3 ó 8:5) son: 320x200, 320x240 –por defecto-, 320x400, 360x240, 376x282, 400x300, 512x384, 640x400, 640x480, 800x600, 1024x768 ó 1280x1024. En el caso de que se escriba una resolución diferente, Bennu mostrará la ventana del juego con ese tamaño que se haya especificado ajustando la resolución a la más próxima de la lista anterior, y si es en modo de pantalla completa, dibujará bordes negros automáticamente alrededor, creando un efecto Cinemascope. Probémoslo:

```
Import "mod_video";

Process Main()
Begin
          set_mode(640,480);
          Loop
                    Frame;
          end
End
```

El tercer parámetro de **set_mode()** puede tener tres valores: el número 8 (o de forma equivalente, la palabra **MODE_8BITS**) , el número 16 (o de forma equivalente, la palabra **MODE_16BITS**), o bien el número 32 (o de forma equivalente, la palabra **MODE_32BITS**). Al ser este parámetro es opcional, si no se escribe nada se sobreentiende, tal como ya hemos comentado anteriormente, que la profundidad será de 32 bits. Este parámetro sirve para indicar la profundidad de color a la que funcionará el juego, así que ninguna imagen cuya profundidad de color sea superior a la establecida por este tercer parámetro no podrá visualizarse correctamente. (consulta el Apéndice B para más información sobre las distintas profundidades de color).

En este manual vamos a trabajar siempre, salvo advertencia expresa, con gráficos de 32 bits, ya que nos aportan más precisión a la hora de definir el color y la posibilidad de usar transparencias. Las otras profundidades sólo tienen sentido ser usadas para tareas muy concretas y especializadas (ya se avisaría en ese caso de su uso en este manual) , o bien en otras arquitecturas diferentes del PC (consolas portátiles ó teléfonos móviles, por ejemplo) con menor potencia de cálculo y menores recursos de memoria, ya que a éstas les puede costar más procesar gráficos de tal profundidad y posiblemente sólo puedan trabajar con imágenes de 16bits, por ejemplo.

*Crear gráficos de 32 bits de profundidad de color no es ningún secreto, se puede hacer con cualquier programa de edición de imágenes. Si usamos Gimp, por ejemplo, basta crear una nueva imagen cuyo espacio de color sea "Color RGB" (que de hecho es la opción por defecto - dentro de "Opciones avanzadas"- del cuadro de diálogo "Crear imagen nueva") y a partir de ahí elegir un color concreto con el que pintar, (seleccionando las componentes de color a partir del cuadro de diálogo correspondiente) y empezar a pintar con la herramienta deseada. Y ya está. No obstante, si usamos el editor Gimp (y otros), a la hora de grabar la imagen en formato Png (que es el único que utilizaremos en este curso) , debemos tener la precaución de vigilar que se grabe realmente en 32 bits y no en 24 (con lo que se perdería la información de transparencia de la imagen). Si en el menú "Capa->Transparencia" aparece la opción activa "Añadir canal alfa", significa que la imagen no tiene el canal alfa y por tanto se guardaría en 24 bits: hay que clicar en esa opción y automáticamente la imagen cambiará a 32 bits, con lo que al grabarla se mantendrá la transparencia.*

*Recuerda, insisto, de guardar la imagen en formato PNG. El Gimp por defecto suele guardar los archivos en un formato especial, el XCF, (su formato nativo) , el cual es muy útil porque guarda la imagen con toda la información necesaria para continuar siendo editada más adelante sin problemas, así que para guardar resultados intermedios que van a volver a ser abiertos en el Gimp es la mejor opción, pero no para grabar la imagen final, porque los gráficos XCF no pueden ser leidos por la mayoría de programas visores de imágenes.*

*Una vez la imagen esté abierta con Gimp en modo 32 bits, incluir zonas transparentes en ella es sencillo. Una manera sería por ejemplo seleccionar un color determinado con la herramienta de selección por color y pulsar seguidamente la tecla "Supr" (o equivalentemente, yendo al menú "Editar->Eliminar"). Se puede jugar con la tolerancia de las herramientas (en nuestro caso la de selección de color, pero también se puede usar con la herramienta borrador,gota,dedo... y otras muchas); la tolerancia es un valor entre 0 y 100 que indica a Gimp qué colores cercanos y parecidos se tendrán en cuenta además del seleccionado. Con ella se pueden conseguir efectos muy interesantes, además de con, por supuesto, la aplicación de los distintos tipos de filtros que incorpora Gimp (como el enfoque/desenfoque, por ejemplo). Por otro lado, otro método alternativo para conseguir convertir un color concreto en transparente es utilizar el cuadro de diálogo que aparece clicando en el menú "Colores->Color a alfa"*

*Si por alguna razón quisieras crear con el Gimp una imagen de 8 bits paletizada (en algún lugar de este manual lo necesitaremos, como por ejemplo en el apartado del seguimiento de caminos con las funciones PATH_*,o en la creación de scrolls Modo7), deberías, antes de guardarla, cambiarla a modo 8 bits, yendo al menú "Imagen->Modo->Indexado" y elegir la paleta deseada en el cuadro de diálogo que aparece. De las múltiples opciones que te ofrece este cuadro, la que escogeremos más habitualmente es la primera, la cual genera una paleta con el número de colores que tú le especifiques. Como máximo se pueden especificar 256 colores y como mínimo el número de colores que tenga la imagen; si eliges un número intermedio se generará una paleta con colores gradualmente intermedios entre los colores existentes en la imagen Una vez generada la imagen de 8 bits paletizada, puedes observar su paleta si vas al menú "Diálogos->Mapa de color", donde podrás editarla mínimamente y sobretodo, podrás comprobar el índice que tiene cada color dentro de ella.*

*Si no estás seguro de si una imagen es de 32, 24 ó los bits que sean, además de cualquier editor de imágenes también puede usar un simple visualizador como XnView (http://www.xnview.com) ó Picasa (http://picasa.google.com) ó similar . Y en Windows es más fácil todavía: tan sólo tienes que clicar con el botón derecho sobre la imagen y seleccionar la opción "Propiedades". En la pestaña "Resumen" (botón "Opciones avanzadas") aparecerá la información que necesitas.*

Por defecto, **set_mode()** inicializa un modo gráfico en ventana. El cuarto parámetro permite especificar un modo gráfico a pantalla completa, además de otros parámetros adicionales. Si este parámetro se especifica –es opcional- , puede ser una combinación (suma) de uno o más de los valores -sin comillas- siguientes:

| | | |
|---|---|---|
| **MODE_FULLSCREEN** | Esta palabra equivale al valor numérico 512 | Crea un modo a pantalla completa. Trabajando a pantalla completa podemos usar cualquier resolución que admita nuestra tarjeta de vídeo y el monitor. De todas maneras, |

| | | |
|---|---|---|
| | | nosotros vamos a trabajar en modo ventana porque si trabajamos en pantalla completa no podremos ver los mensajes generados por los errores que cometemos, por ejemplo. |
| **MODE_WINDOW** | Esta palabra equivale al valor numérico 0 | Crea un modo en ventana –por defecto-. |
| **MODE_2XSCALE** | Esta palabra equivale al valor numérico 256 | Dobla la resolución gráfica de nuestro juego. Es decir, si el modo de vídeo fuera 320x200, la pantalla tendrá un tamaño de 640x400. Los bordes de los gráficos aparecerán suavizados. |
| **MODE_HARDWARE** | Esta palabra equivale al valor numérico 2048 | Crea una superficie en buffer de vídeo para la ventana o pantalla. Es decir, normalmente, Bennu dibuja sobre la memoria RAM del ordenador y copia el resultado en la memoria de la tarjeta gráfica; con este parámetro activado, Bennu dibujará directamente sobre la memoria de la tarjeta de vídeo. Esto acelera algunas operaciones, pero enlentece otras (como por ejemplo, el dibujo con transparencias). |
| **MODE_DOUBLEBUFFER** | Esta palabra equivale al valor numérico 1024 | Habilita un segundo buffer; y en cada fotograma se utiliza uno u otro alternativamente. Su interés suele venir en combinacion con MODE_HARDWARE. Puede conseguir que las transiciones entre fotogramas sean más suaves, pero se pierde la compatibilidad *restore_type=partial_restore* |
| **MODE_MODAL** | Esta palabra equivale al valor numérico 4096 | Hace que la ventana se abra en modo MODAL, evitando que la ventana pierda el foco y confinando el movimiento del raton a su superficie. |
| **MODE_FRAMELESS** | Esta palabra equivale al valor numérico 8192 | Crea una ventana sin bordes. |
| **MODE_WAITVSYNC** | Esta palabra equivale al valor numérico 16384 | Modo de vídeo que espera a la sincronización vertical del redibujado, para evitar el efecto gelatinoso |

Por ejemplo, el siguiente código hace uso del cuarto parámetro de **set_mode()**. Si lo ejecutas verás que este ejemplo pondrá la ventana del programa inicialmente a pantalla completa, al cabo de un segundo cambiará la resolución y pondrá la ventana "normal", seguidamente volverá a cambiar la resolución y pondrá la ventana sin marcos, y finalmente, volverá a cambiar la resolución y pondrá la ventana en modo 2xscale:

```
Import "mod_video";

Process Main()
begin
//Pronto se explicará para qué sirve esta función.Para entender el ejemplo no es importante.
  set_fps(1,1);
  set_mode(640,480,32, mode_fullscreen);
  frame;
  set_mode(320,240,32, mode_window);
  frame;
  set_mode(800,600,32, mode_frameless);
  frame;
  set_mode(640,480,32, mode_2xscale);
  frame;
  loop
    frame;
  end
end
```

Si la función **set_mode()** crea una ventana de juego, ésta utilizará por título el nombre del fichero DCB y un icono estándar. Pero eso se puede cambiar. La función **set_title()** establece el título de la ventana del juego (el cual se pasa entre comillas como el único parámetro que tiene), y la función **set_icon()** usa un gráfico determinado como icono para mostrar en la barra de título de la ventana del juego. El funcionamiento de ambas funciones lo estudiaremos posteriormente en este mismo capítulo, cuando hayamos introducido el uso de gráficos. Pero ya indicamos ahora que es recomendable siempre usar estas funciones (**set_title()** y **set_icon()**) inmediatamente antes de cada llamada a

**set_mode()** que se haga en el programa, ya que si después de estas funciones no aparece la función **set_mode()**, no se visualizarán sus efectos.

A modo de síntesis, pues, si por ejemplo quisiéramos hacer que nuestro juego funcionara a 800x600 de resolución, con profundidad de 16 bits, con ventana modal sin bordes, tendríamos que escribir lo siguiente:

```
Import "mod_video";

Process Main()
begin
          Set_mode(640,480,MODE_16BITS,MODE_MODAL + MODE_FRAMELESS);
          Loop
                    Frame;
          end
end
```

Como alternativa al operador "+" para unir y así poder utilizar varias opciones a la vez en el cuarto parámetro, se puede utilizar el operador "|": se obtiene el mismo resultado.

Si queremos que se vea un título, hemos de hacer que se vean los bordes:

```
Import "mod_video";

Process Main()
begin
           Set_title("Prueba");
          Set_mode(640,480,MODE_16BITS,MODE_MODAL);
          Loop
                    Frame;
          end
end
```

## *Configuración de los frames per second (FPS). Orden "Set_fps()"

Lo siguiente que haremos después de establecer el modo de pantalla con **set_mode()**, será definir el número de imágenes por segundo que vamos a visualizar. Ya he comentado que cualquier animación o imagen en movimiento se consigue haciendo dibujos similares con pequeñas diferencias, por ejemplo, de posición. Esto es lo que se hace en la televisión, se muestran imágenes fijas a mucha velocidad, una tal que el ojo humano no pueda diferenciarlo. Pues nosotros podemos decidir cuantas imágenes por segundo vamos a ver, o como se dice en inglés, los "frames per second" (FPS).

Si has jugado mucho tiempo con videoconsolas, ya habrás oído aquello de juegos a 50Hz o a 60Hz (herzios) que no es más que la medida de imágenes por segundo que es capaz de "pintar" la televisión, y ciertamente hablamos de valores muy elevados. Se ha demostrado que el ojo humano no es capaz de distinguir imágenes fijas a más de 32 imágenes por segundo, pero realmente, si tienes buena vista, puedes notar que la imagen va a saltos, y a mayor número de imágenes por segundo más suave se verá el juego,(y por supuesto menos perjudicial para la vista); pero en el otro extremo tenemos los requisitos del sistema: un ordenador necesita hacer cientos de operaciones entre imagen e imagen, como calcular la posición de los objetos, "pintar" la pantalla, manejar inteligencia artificial y demás, y para ello es necesario dejarle el tiempo suficiente. Por eso, cuantas más imágenes por segundo tengamos, menos tiempo tiene el ordenador para realizar todas las operaciones y más rápido debe ser, de lo contrario, el juego se detendría hasta que se hubieran completado todas las operaciones: es lo que normalmente se le llama "ralentización". Así que es importante escoger un buen "frame rate" que equilibre la balanza.

A la hora de diseñar un videojuego es muy importante tener presente en qué tipo de ordenadores se va a poder jugar: si la potencia no es un problema podemos poner un número elevado de imágenes por segundo, pero si es para ordenadores más modestos (o dispositivos móviles) hay que reducir ese valor, incluso sacrificar algo de suavidad en los movimientos a favor de la velocidad general del juego. Yo recomiendo que, a la hora de valorar la situación, que el fps se mantenga en unos valores entre 30 y 60 imágenes por segundo, aunque realmente, con las máquinas que hay ahora, los juegos que vas a realizar y el rendimiento general de Bennu, es fácil que puedas usar los 60 fps en casi todos

tus juegos sin problemas.

Para especificar un valor de fps, se utiliza la función **set_fps(),** definida también en el módulo "mod_video". Si escribimos por ejemplo *Set_fps(60,1);*, lo que estaremos diciendo al ordenador es que trabaje a 60 imágenes por segundo.

Quizás te llame la atención el segundo parámetro. Ese 1 es algo que puede ayudar al rendimiento, porque le da permiso al ordenador literalmente a "saltarse 1 frame". Es decir, le dice al ordenador que en cada segundo, si le falta tiempo, puede ahorrarse una vez el "dibujar" en la pantalla, lo que puede provocar un salto en la secuencia de la imagen. Visualmente esto no es aceptable, pero darle un cierto margen al ordenador para que termine su trabajo a tiempo es algo más que recomendable, sobre todo si los equipos que van a ejecutar tu juego son más bien modestos. La cifra puede variar entre 0 (sin saltos) y el número de saltos por segundo que tú quieras: prueba la cifra que más se ajuste a tus necesidades y resultados deseados.

Como nota adicional, has de saber que si especificamos un valor de 0 fps, le estamos diciendo al ordenador que ejecute el juego tan rápido como sea posible, de tal manera que cuanto más rápido sea el ordenador más ligero irá el juego. Esta utilidad está especialmente recomendada para programas que deben mostrar resultados en pantalla cuanto más rápido mejor, como dibujar una gráfica o contar bucles, pero si se usa para juegos piensa en como irá en un Pentium a 133Mhz y luego en un Pentium Dual Core a 3.7GHz.

Para jugar un poco con esta función, recuperemos el primer programa que hicimos, el del scroll horizontal de un texto:

```
Import "mod_text";

Process Main()
Private
          int mivar1;
End
Begin
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end
end
```

Vemos que aquí no hay ninguna función **set_fps()**. Si no aparece, se toma por defecto el valor de 25 fps. Añadamos la orden, con un valor de 60 fps. ¿Qué es lo que ocurrirá?:

```
Import "mod_text";
Import "mod_video";

Process Main()
Private
          int mivar1;
End
Begin
           set_fps(60,1);
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end
end
```

El scroll se mueve más rápido. Esto es lógico: estamos obligando a que se pinten más fotogramas en cada segundo. Es decir,estamos forzando a que el ritmo de ejecución del programa sea tal que la orden frame; se ejecute 60 veces por segundo, o al menos 59. Es otra manera de decir que **set_fps()** establece la velocidad del juego. Es fácil ver esto si ponemos que se vea un frame cada segundo:

```
Import "mod_text";
Import "mod_video";

Process Main()
Private
          int mivar1;
End
Begin
           set_fps(1,1);
          mivar1=10;
          while(mivar1<320)
                    delete_text(0);
                    mivar1=mivar1+2;
                    write(0,mivar1,100,1,"¡Hola mundo!");
                    frame;
          end
end
```

Vemos que el texto se mueve en cada segundo, porque es la frecuencia a la que se ejecuta FRAME.

La función **set_fps()** la puedes invocar en cualquier punto de tu código, con lo que podrás alterar la frecuencia de frames por segundo según convenga (si el personaje está en una fase u otra del juego,etc).

Existe una manera de visualizar en todo momento a cuántos FPS funciona un programa mientras éste se está ejecutando (ya que este valor, según lo que hayamos puesto como segundo parámetro de **set_fps()**, puede ir variando). Esto es interesante para detectar cuellos de botella en determinados momentos de la ejecución de un programa debido al uso excesivo de recursos de la máquina (memoria, disco, CPU...) por parte de éste. Simplemente tienes que poner en cualquier sitio entre el BEGIN/END del programa principal -aunque se recomienda al principio de todo, justo después de **set_fps()**-:

```
write_var(0,100,100,4,fps);
```

Lo que estamos haciendo simplemente es imprimir por pantalla el valor actualizado de una variable, llamada obligatoriamente *FPS*, que nos marcará precisamente lo que queremos: a cada frame nos dirá a cuántos fps está funcionando nuestro programa. Esta variable NO LA TIENES que declarar: ya viene declarada por defecto en Bennu: es decir, ya está lista para usar (dentro de **write_var()** como hemos puesto o en cualquier otro sitio) y no tienes que hacer nada más. Esta variable es una (entre otras muchas) de las llamadas "variables predefinidas" de Bennu, y en concreto, es una variable predefinida global. Tranquilo:sobre los tipos de variables ya hablaremos en el capítulo siguiente.

Existen más variables predefinidas de Bennu bastante similares a *FPS*. Una de ellas es la variable predefinida global *FRAME_TIME*, la cual es de tipo FLOAT. Imprimiendo por pantalla su valor a cada frame podremos observar el tiempo que ha pasado desde el último frame anterior; es decir,esta variable muestra la diferencia entre el principo del último frame y el frame actual. Haciendo un poco de cálculo se puede demostrar que *FPS=1/FRAME_TIME*. Un ejemplo de su uso puede ser el siguiente, donde

```
import "mod_text";
import "mod_key";
import "mod_video";

process main()
Private
    int i;
    float f;
End
Begin
```

```
    set_fps(40,1);
    write_var(0,0,0,0,frame_time);
    write_var(0,0,10,0,i);
    write_var(0,0,20,0,f);
    Repeat
//Calcula cuánto tiempo ha estado el programa ejecutándose con decimales
          f = f + frame_time;
//Calcula cuánto tiempo ha estado el programa ejecutándose sin decimales
         i = f;
         frame;
   Until(key(_ESC))
End
```

Otra variable global predefinida relacionada con los fotogramas es *SPEED_GAUGE*. Su valor el tanto por ciento de coincidencia entre los FPS reales del programa en un instante determinado y el valor teórico especificado por **set_fps()**. Un valor de 100 significa que el número de fotogramas por segundo que realmente se están visualizando son los que **set_fps()** efectivamente marca. Un valor inferior a 100 indica que los FPS reales no llegan al número indicado idealmente. Es decir, se podría considerar que *speed_gauge=(FPSreales/FPSteoricos)*100*. Un ejemplo:

```
import "mod_text";
import "mod_key";
import "mod_video";

process main()
Begin
    set_fps(140,1);
    write_var(0,0,0,0,fps);
    write_var(0,0,10,0,speed_gauge);
    Repeat
        frame;
    Until(key(_ESC))
End
```

## *Concepto de FPG. Editores de FPGs

Todavía no hemos pintado ningún gráfico, y ya es hora. Pero antes de nada, hay que disponer de los gráficos a utilizar. Éstos se pueden crear,ya lo hemos visto, de múltiples maneras. Aconsejo utilizar el Inkscape para creación de gráficos vectoriales muy eficaces o el Gimp para mapas de bits. Si no los tienes instalado, puedes usar cualquier otro programa. A malas, siempre podrás usar el limitado MSPaint, que ya es suficiente para hacer nuestras pruebas.

Una vez que tenemos los gráficos, ¿cómo los incluimos en el código de nuestro programa para que se muestren? Hay dos maneras: o referenciar uno a uno cada gráfico que utilicemos en el juego de forma independiente, o haber generado antes un fichero único que sirva de paquete contenedor de los gráficos, de manera que solamente se referencie en el código ese archivo, a partir del cual se pueden localizar los gráficos individuales. Este archivo contenedor ha de tener la extensión FPG. Así que el FPG es como un cajón, un almacén donde se guardan todos los graficos. Ambas soluciones son válidas, aunque cuando el número de gráficos empieza a ser elevado, es conveniente irlos empaquetando en unos pocos FPG dependiendo de su función específica dentro del juego, para tener una clasificación más o menos clara de los gráficos: así, tendríamos los archivos "fondos.fpg", "enemigos.fpg",etc conteniendo los distintos gráficos de fondos, enemigos,etc respectivamente. Además hay que tener en cuenta que los ficheros FPG pueden ser de 8, 16 ó 32 bits también: esto depende del tipo de imágenes que van a incluir en ellos. No se pueden mezclar imágenes de distintas profundidades de color en un mismo FPG: se deberán de empaquetar por separado en diferentes FPGs, cada uno con su profundidad correspondiente. Y, evidentemente, un FPG que tenga menor profundidad que alguna de las imágenes que incluya será una fuente constante de errores.

La pregunta que surge ahora es: si ya se tienen los gráficos, y si quieren utilizar ficheros fpg, ¿cómo se crean éstos? Tal como se detalló en el capítulo anterior, existen varias aplicaciones que realizan este proceso, e incluso se la puede programar uno mismo con Bennu (lo veremos más adelante), pero tal vez, de todas las que nombramos en

la lista del capítulo anterior, la más popular sea "FPGEdit" (aunque sólo funciona en Windows), que además viene incluida en el BennuPack.

El "FPGEdit" es muy sencillo de utilizar: basta recorrer el disco duro mediante el árbol que aparece en la zona superior izquierda de la ventana del programa en busca de las imágenes que queremos incluir en el FPG. A medida que nos movamos por las diferentes carpetas, irán apareciendo listadas las imágenes incluidas en su interior, en la parte superior derecha de la ventana del programa.Seguidamente, hemos de decirle que queremos crear un nuevo fichero FPG e introducir en él los gráficos que deseemos.Esto se hace utilizando la parte inferior de la ventana del programa, donde crearemos el FPG. En el cuadro que aparece cuando queremos crear un FPG nuevo, escogeremos la ruta donde se grabará el archivo FPG con el nombre que queramos y como tipo de FPG el de 32 bits de Bennu (¡esto es muy importante!) y a partir de ahí iremos seleccionando desde la zona superior derecha de la ventana los gráficos que deseamos incluir (se pueden seleccionar varios manteniendo pulsada la tecla CTRL) , arrastrándolos y soltándolos en la zona inferior de la ventana. Fíjate que cada gráfico, una vez incluido en el FPG, tendrá asociado un número de tres cifras. Este número será el identificador único de ese gráfico en aquel código fuente donde se utilice el FPG. Es decir, que cuando se quiera hacer referencia a ese gráfico en el código, aparte de decir en qué FPG está incluido, habrá que decir cuál es el número concreto que corresponde al gráfico buscado. Como podrás deducir, “sólo” caben hasta 999 gráficos en un FPG (el 000 no vale), y puedes comprobar que ese código lo puedes cambiar según tus intereses (con el botón "Edit" de la barra superior de la zona inferior de la ventana), siempre que mantengas que es un número único para ese FPG. Una vez hayamos seleccionado los gráficos que queríamos, hemos de guardar el fichero FPG.Y siempre podremos abrir un archivo FPG ya creado para editarlo: quitar o poner nuevas imágenes o cambiar sus códigos identificativos,etc.

Otro editor FPG (para Windows también) que comentamos en el capítulo anterior es el "Smart FPG Editor". Es más sencillo si cabe que el FPGEdit. Tal sólo consta de una sección inferior donde se muestran las imágenes contenidas en el FPG seleccionado en cada momento, y una barra de botones que realizan las acciones básicas: crear FPGs de 8, 16 y 32 bits, duplicar FPGs, añadir ó sobrescribir imágenes Png al FPG (especificando su identificador y sus posibles puntos de control), eliminar imágenes del FPG y extraer imágenes del FPG. Lo justo y necesario para un editor pequeño pero funcional.

Otro editor FPG (esta vez multiplataforma, ya que está programado con el propio Bennu) es el de Prg. Es ligeramente más complejo que los anteriores, aunque también es verdad que aporta más funcionalidad y no sólo es editor de FPGs: también es capaz de editar fuentes en formato FNT y trabajar con imágenes en tres dimensiones. Recomiendo probarlo para descubrir sus múltiples opciones.

## *Carga de la imagen de fondo. Órdenes "Load_png()", "Load_fpg()", "Put_Screen()", "Clear_Screen()"

Lo primero que vamos a hacer es utilizar una imagen PNG como imagen de fondo de pantalla.Como acabamos de decir, siempre tendremos la posibilidad de utilizar o bien imágenes individuales o bien imágenes incluidas dentro del contenedor FPG. En los ejemplos siguientes utilizaremos los dos sistemas para que los compares, pero posteriormente utilizaremos uno u otro indistintamente.

Una vez que tengas creada la imagen de 32 bits (siempre será así si no se dice lo contrario) de fondo, y de hecho, todas las imágenes que vayas a utilizar en el juego, lo más cómodo es que la/s guardes en la misma carpeta donde tienes el PRG que estarás haciendo. Así no habrá problemas de rutas y se encontrarán las imágenes inmediatamente. Si se supone que nuestra imagen PNG de fondo se llama “fondo.png", escribe el siguiente código:

```
Import "mod_video";
Import "mod_screen";
Import "mod_map";

Process Main()
private
      int id1;
end
Begin
      set_mode(600,400,32);
      id1=load_png("fondo.png");
      put_screen(0,id1);
      loop
```

```
            frame;
        end
end
```

Lo novedoso empieza en la línea *id1=load_png("fondo.png");*. La función **load_png()** -definida en el módulo "mod_map"- sólo tiene un parámetro que es el nombre de la imagen que queremos cargar –de ahí su nombre- en memoria RAM, posibilitando así su manipulación posterior por parte del programa. Es importante saber que cualquier imagen o sonido o video con el que deseemos trabajar en nuestro código siempre tendremos que cargarlo previamente desde el disco duro a la RAM, mediante una función load_XXX –dependiendo de lo que carguemos-; esto es una condición indispensable. De hecho, de aquí viene la importancia de tener una memoria RAM lo más grande posible: para que quepan cuantas más imágenes, sonidos y videos mejor. Y también hay que tener claro que cargar una imagen en memoria no significa que se vaya a ver nada de nada: la carga es un proceso necesario pero que es interno del programa, en la pantalla no se va a ver nada.

Si la imagen no estuviera en la misma carpeta que el archivo DCB, tendríamos que poner como parámetro la ruta de la imagen a partir del lugar donde está el archivo DCB, en vez de sólo el nombre. Es decir, que si tenemos por ejemplo en Windows una carpeta "Juego" que contiene tanto el archivo DCB como una carpeta llamada "imágenes" donde está a su vez nuestro "fondo.png" deberíamos poner

```
id1=load_png("imagenes\fondo.png");
```

o bien, de forma alternativa, poner toda la ruta desde la unidad donde estemos –la ruta absoluta que se dice-.Si tenemos por ejemplo la carpeta "Juego" en la raiz C:, pues sería:

```
id1=load_png("C:\Juego\imagenes\fondo.png");
```

Puedes observar que esta función devuelve un valor entero, que se lo asignamos a una variable creada por nosotros llamada *id1*. Este valor entero es un identificador único, que nos servirá para referenciar en cualquier momento del código a la imagen que hemos cargado. Es decir, a partir de ahora, para manipular y gestionar la imagen "fondo.png" nos referiremos a ella mediante esa variable *id1* que contendrá el "DNI" particular e instransferible de dicha imagen: en vez de nombrarla cuando sea necesario con su nombre, la nombraremos con ese número identificador, (el cual pocas veces nos interesará saber cuál es en concreto y simplemente lo asignaremos a la variable entera con la que se trabajará).

Puedes ver también que rápidamente utilizamos la variable id1 porque en la línea siguiente aparece la función **put_screen()** -la cual se define en el módulo "mod_screen"- con dos parámetros, que valen 0 e *id1* respectivamente. Esta función lo que hace es pintar sobre el fondo un gráfico que no cambia ni se mueve. Ese gráfico viene dado por el segundo parámetro, el cual es el valor numérico identificativo del PNG que acabamos de cargar. El primer parámetro siempre ha de valer 0 si lo que queremos es visualizar una imagen cargada individualmente, tal como hemos hecho; si hubiéramos cargado un archivo FPG –ahora lo veremos-, este primer parámetro ha de valer otra cosa.

La función **put_screen()** no necesita que se le indique la posición del gráfico, pues este siempre se pondrá en el centro de la pantalla, independientemente de la resolución, por eso normalmente la imagen usada con esta función ocupa toda la pantalla o gran parte de ella. Si no llegara a ocupar toda la pantalla lo que falta se rellena en negro.

Ya que estamos hablando de la función **put_screen()**, aprovecho para comentar la existencia de la función **clear_screen(),** definida también en el módulo "mod_screen". Esta función, que no tiene ningún parámetro, borra el gráfico del fondo de la pantalla. Puedes comprobarlo ejecutando este pequeño ejemplo:

```
Import "mod_video";
Import "mod_screen";
Import "mod_map";

Process Main()
private
 int id1;
end
Begin
 set_mode(600,400,32);
 set_fps(1,0);
 id1=load_png("fondo.png");
 put_screen(0,id1);
```

```
 frame;
 clear_screen();
 frame;
 put_screen(0,id1);
 frame;
 clear_screen();
 frame;
 put_screen(0,id1);
 loop
     frame;
 end
end
```

Ahora vamos a hacer lo mismo que veníamos haciendo hasta ahora -es decir, poner un gráfico de fondo y ya está-,pero en vez de cargar la imagen individualmente con **load_png()**, vamos a incluirla dentro de un fichero FPG –que de momento sólo contendrá esta imagen-, y lo que vamos a hacer es cargar el fichero FPG, con lo que todas las imágenes que contuviera quedarían cargadas automáticamente y no se tendría que ir cargándolas una a una.

Lo primero que hay que hacer es crear el archivo FPG con cualquier editor FPG de los ya conocidos.Lo nombraremos "imágenes.fpg" e incluiremos "fondo.png" con el código 001. Seguidamente, escribimos este programa:

```
Import "mod_video";
Import "mod_screen";
Import "mod_map";

Process Main()
Private
       int id2;
end
Begin
       set_mode(640,480,32);
       id2=load_fpg("imagenes.fpg");
       put_screen(id2,1);
       Loop
           Frame;
       End
end
```

Lo que hacemos aquí es utilizar la función **load_fpg()** -definida en el módulo "mod_map"-, la cual es equivalente a **load_png()**, pero en vez de cargar en memoria un archivo PNG, carga un archivo FPG. Sólo a partir de estonces podrán estar accesibles en el código las imágenes que contiene. Igual que antes, esta función devuelve un número entero –aleatorio-, que se recoge en la variable *id2*, que identificará al archivo FPG unívocamente a lo largo del programa para lo que sea necesario. Y como antes también, cargar un archivo de imágenes no quiere decir que se vea nada de nada. Para ello hay que decirle más cosas al ordenador. Si queremos por ejemplo, volver a poner "fondo.png" como imagen de fondo estática, hemos de volver pues a utilizar la función **put_screen()**.

La diferencia con el ejemplo anterior está en que ahora el primer parámetro no vale 0 sino el identificador devuelto por **load_fpg()**. Siempre que carguemos el FPG, cuando queramos hacer referencia a alguna de las imágenes que contiene, tendremos primero que indicar en qué FPG se encuentra, y posteriormente decir qué código de imagen tiene dentro de él. Es lo que hacemos aquí: decimos que queremos cargar una imagen que está en el fichero identificado por *id2* cuyo código interno es 001 (el valor del segundo parámetro). Es por esta razón que cuando cargamos las imágenes directamente, el primer parámetro vale 0, porque no necesitamos referenciar a ningún FPG.

## *Descarga de imágenes: órdenes "Unload_map()" y "Unload_fpg()". Orden "Key()"

Acabamos de decir que para poder utilizar imágenes en nuestro juego hay que cargarlas previamente en memoria RAM. Pero la RAM es finita. Si empezáramos a cargar y cargar imágenes sin control, al final no tendríamos suficiente RAM disponible y nuestro juego se colgaría. Para evitar esto, es necesario descargar las imágenes que ya no estén siendo utilizadas, para dejar libre espacio de RAM que pueda ser usado otra vez.

Para descargar una imagen (cargada previamente de manera individual con **load_png()**, o bien formando parte de un FPG cargado previamente con **load_fpg()**) no existe una función "unload_png()" que realice esta tarea, sino que se llama **unload_map()** (veremos más adelante por qué se llama así). Esta función está definida, al igual que **load_png(),** en el módulo "mod_map" y tiene dos parámetros. El primero representa el número de FPG donde estaría incluida la imagen a descargar, (si se cargó de forma individual con **load_png()** no existe ningún FPG y por tanto su valor será 0. El segundo parámetro representa el código de la imagen en el interior del FPG si éste existiera, o si no, el identificador de la imagen devuelto por **load_png()** cuando ésta se cargó.

Para descargar una FPG entero cargado previamente mediante **load_fpg(),** disponemos de la función **unload_fpg()**, también definida en el módulo "mod_map". Esta función tiene un único parámetro que es el identificador del FPG que es devuelto por **load_fpg()** a la hora de cargarlo. Hay que tener en cuenta que si se descarga un FPG, ninguna de las imágenes incluidas en él se podrán utilizar en el programa, por lo que hay que estar seguro de que ninguna imagen del FPG será ya más utilizada.

Utilizaremos el ejemplo que acabamos de hacer en el apartado anterior para aprender por ejemplo cómo se descargan todas las imágenes de un FPG, usando **unload_fpg().** Sin embargo, antes de eso, modificaremos dicho ejemplo para que realice otra cosa que ya va siendo hora de que nuestros programas lo hagan por fin: que sus ventanas puedan ser cerradas de una vez de forma correcta y sin que salten errores del sistema. Así que primero haremos la siguiente modificación:

```
Import "mod_video";
Import "mod_screen";
Import "mod_map";
Import "mod_key";

Process Main()
Private
       int id2;
end
Begin
       set_mode(640,480,32);
       id2=load_fpg("imagenes.fpg");
       put_screen(id2,1);
       Loop
           if (key(_esc))
                  break;
           end
           Frame;
       End
end
```

Fíjate que lo único que hemos hecho ha sido añadir el siguiente if:

```
if (key(_esc))
       break;
end
```

(Este if también se podría haber escrito en una sola línea, así: *if(key(_esc)) break; end* , aunque no es lo recomendado en la "Guía de Estilo". De hecho, el salto de línea en el código fuente, Bennu no lo tiene en cuenta nunca para nada: es como si no estuviera).

Lo que dice este if es que si es cierta la condición que contiene entre paréntesis (ahora veremos qué es), se ha de salir del LOOP. Al salir del LOOP, de hecho se acaba el programa. Por lo que la condición del if que hemos puesto sirve para salir del programa en el momento que *key(_esc)* sea cierto. Mientras que no lo sea, el if no se ejecutará y el LOOP irá funcionando.

¿Y qué quiere decir esto de *key(_esc)*? Pues **key()** es una función definida en el módulo "mod_key",que devuelve "verdadero" (es decir, 1) o "falso" (es decir, 0) si se ha pulsado una determinada tecla del teclado (que hayamos especificado), o no, respectivamente. Por lo tanto, lo que hace el if es comprobar, a cada iteración de LOOP, si se ha pulsado la tecla ESC. Si no se ha pulsado, el if no se hace (ya que la condición es "falsa": **key()** devuelve 0 y para el if el 0 siempre es igual a "falso") y se ejecuta FRAME y el programa continúa. Si llega un momento en que sí se ha pulsado ESC, entonces la condición *key(_esc)* es verdadera (porque en ese momento **key()** devuelve 1), se ejecuta el interior del if y por tanto se sale del bucle y se acaba el programa. ¡Acabamos de descubrir una nueva forma de acabar

el programa que no sea dándole al botón de Cerrar de la ventana!: pulsando una tecla determinada podemos hacer lo mismo.

Evidentemente, la función **key()** no sólo funciona con la tecla ESC. Aquí tienes la lista de posibles teclas (son fáciles de deducir) que la función **key()** puede tratar. Como puedes ver, en realidad cada uno de estos valores equivale a un número, el cual se podría utilizar también de forma alternativa con **key()**, así,por ejemplo *key(_z)* sería lo mismo que *key(44)*.

| | | |
|---|---|---|
| _ESC = 1 | _F = 33 | _F7 = 65 |
| _1 = 2 | _G = 34 | _F8 = 66 |
| _2 = 3 | _H = 35 | _F9 = 67 |
| _3 = 4 | _J = 36 | _F10 = 68 |
| _4 = 5 | _K = 37 | _NUM_LOCK = 69 |
| _5 = 6 | _L = 38 | _SCROLL_LOCK = 70 |
| _6 = 7 | _SEMICOLON = 39 | _HOME = 71 |
| _7 = 8 | _APOSTROPHE = 40 | _UP = 72 |
| _8 = 9 | _WAVE = 41 | _PGUP = 73 |
| _9 = 10 | _L_SHIFT = 42 | _C_MINUS = 74 |
| _0 = 11 | _BACKSLASH = 43 | _LEFT = 75 |
| _MINUS = 12 | _Z = 44 | _C_CENTER = 76 |
| _PLUS = 13 | _X = 45 | _RIGHT = 77 |
| _BACKSPACE = 14 | _C = 46 | _C_PLUS = 78 |
| _TAB = 15 | _V = 47 | _END = 79 |
| _Q = 16 | _B = 48 | _DOWN = 80 |
| _W = 17 | _N = 49 | _PGDN = 81 |
| _E = 18 | _M = 50 | _INS = 82 |
| _R = 19 | _COMMA = 51 | _DEL = 83 |
| _T = 20 | _POINT = 52 | _F11 = 87 |
| _Y = 21 | _SLASH = 53 | _F12 = 88 |
| _U = 22 | _R_SHIFT = 54 | _LESS = 89 |
| _I = 23 | _PRN_SCR = 55 | _EQUALS = 90 |
| _O = 24 | _ALT = 56 | _GREATER = 91 |
| _P = 25 | _SPACE = 57 | _ASTERISK = 92 |
| _L_BRACHET = 26 | _CAPS_LOCK = 58 | _R_ALT = 93 |
| _R_BRACHET = 27 | _F1 = 59 | _R_CONTROL = 94 |
| _ENTER = 28 | _F2 = 60 | _L_ALT = 95 |
| _CONTROL = 29 | _F3 = 61 | _L_CONTROL = 96 |
| _A = 30 | _F4 = 62 | _MENU = 97 |
| _S = 31 | _F5 = 63 | _L_WINDOWS = 98 |
| _D = 32 | _F6 = 64 | _R_WINDOWS = 99 |

¿Y por qué explico todo este rollo? Porque con el añadido del if, podemos parar el programa por código cuando deseemos. Y justo en ese momento, antes de salir de la ejecución, insertaremos la función de descarga de FPGs **unload_fpg()**, para acabar el programa limpiamente sin dejar ningún residuo en la memoria RAM.

```
Import "mod_video";
Import "mod_screen";
Import "mod_map";
Import "mod_key";

Process Main()
Private
       int id2;
end
Begin
       set_mode(640,480,32);
      id2=load_fpg("imagenes.fpg");
      put_screen(id2,1);
      Loop
          if (key(_esc))
                 break;
          end
```

```
            Frame;
        End
        unload_fpg(id2);
end
```

Se ve que una vez salido del bucle, y justo antes de acabar el programa se descarga el FPG que hayamos utilizado. Fijarse que para ello,la función **unload_fpg()** requiere que el valor de su único parámetro sea el identificador del fichero FPG que se quiere descargar entero. Y ya está.

Probaremos otro ejemplo, esta vez mostrando cómo se descarga una imagen suelta, con **unload_map()**

```
Import "mod_video";
Import "mod_map";
Import "mod_screen";
import "mod_key";
import "mod_text";

Process Main()
Private
            Int mipng;
End
Begin
            mipng=load_png("fondo.png");
            write_var(0,100,100,4,mipng);
            set_mode(640,480,32);
            put_screen(0,mipng);
            Loop
            if(key(_esc)) break; end
                        Frame;
            end
            unload_map(0,mipng);
            clear_screen();
            put_screen(0,mipng);
            while (!key(_space)) frame; end
End
```

Este ejemplo tiene detalles muy interesantes. Primero cargamos un PNG individual y mostramos el identificador que nos devuelve **load_png().** Podemos comprobar cómo éste tendrá el valor de 1000 siempre: los identificadores de las imágenes individuales empiezan a partir de ese número. Seguidamente establecemos el modo de vídeo y colocamos la imagen como fondo. A partir de ahí, entramos en el bucle del cual no saldremos hasta que no pulsemos la tecla "ESC". Mientras no salgamos, veremos la imagen de fondo. Lo interesante llega ahora: mediante **unload_map()** descargamos la imagen PNG de la memoria, pero atención, necesitamos de **clear_screen()** para que dicha imagen se deje de ver de la pantalla. Esto es muy importante: a pesar de haber descargado la imagen de memoria, todas las muestras de dicha imagen que se pintaron en pantalla,continuarán allí, porque aunque la imagen como tal deje de ser accesible, su rastro permanece visible: puedes comprobarlo si comentar la línea **clear_screen().** Finalmente, volvemos a intentar colocar la misma imagen como fondo otra vez mediante otro **put_screen()**, pero ahora, evidentemente, no lo podremos conseguir porque la imagen ya no está disponible, y no veremos nada. Puedes comprobar también que el identificador permanece inalterable durante toda la ejecución del programa, aunque la imagen asociada a él haya sido descargada: el identificador es una simple variable a la que se le asigna un determinado valor, y mientras no se le asigne otro diferente, seguirá guardando el que adquirió en su momento, independientemente de lo que ocurra o deje de ocurrir con las imágenes asociadas. Sólo concluir diciendo que el último bucle es para que el programa pueda seguir ejecutándose hasta que se pulse la tecla "Space".

Prueba por último este otro código curioso. A ver si sabes qué es lo que ocurre; (necesitarás para poderlo ejecutar un archivo FPG con cuatro imágenes en su interior como mínimo).

```
Import "mod_video";
Import "mod_screen";
Import "mod_map";
Import "mod_rand";

Process Main()
private
    int id1;
```

```
end
begin
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    loop
          put_screen(id1,rand(1,4));
          frame;
    end
end
```

## *Mapas

¿Por qué no existe una función llamada "unload_png()" y debemos recurrir a **unload_map()** para descargar imágenes de la memoria RAM? Porque cuando Bennu carga una imagen, realiza en memoria una transformación de su formato (sin que el usuario perciba nada) y convierte dicha imagen en un formato propio de Bennu que es el formato MAP. De hecho, por "mapa" se entiende todo aquél gráfico que ha sido cargado por Bennu y que en ese momento está en memoria.

Antes de ser cargados, los mapas pueden ser cualquier tipo de imagen: Bennu por sí mismo es capaz de cargar imágenes PNG y PCX -es otro formato- pero con las librerías adecuadas podría ser capaz de cargar imágenes en multitud de otros formatos (BMP, JPG, GIF, TIFF...). Independientemente del formato de cada una de estas imágenes en disco, cuando Bennu las carga las reconvierte invariablemente en su formato nativo, MAP. De esta manera, Bennu siempre trabaja de la misma forma con todas las imágenes, provengan de donde y como provengan. Para Bennu sólo existe un formato de imagen, que es MAP, y todas las manipulaciones que haga las hará sobre ese formato. Por eso no existe "unload_png()": porque para Bennu lo que hay en memoria no es una imagen PNG, sino un mapa. Dicho de otra forma, todas las imágenes, tengan el formato en disco que tengan, se descargarán de la misma manera, con **unload_map()**, porque todas tienen el mismo formato en memoria.

Bennu ofrece la posibilidad de grabar en disco la imagen en el mismo formato MAP con el que está en ese momento cargada en memoria. Para ello dispone de la función **save_map()**. Y lógicamente, Bennu también incluye otra función que permite realizar el proceso inverso: cargar desde disco imágenes previamente guardadas en formato MAP: **load_map()**. De hecho, un contenedor FPG que cargamos con **load_fpg()** no es más que una colección de imágenes MAP (hasta un máximo de 1000) guardadas en disco. No obstante, en este texto no utilizaremos ni **save_map()** ni **load_map()**: en disco siempre trabajaremos con las imágenes en formato PNG, puesto que este formato ya nos aporta todas las características que necesitamos y más.

## *Iconos. Órdenes "Set_icon()" y "Set_title()". Editores de recursos

Un icono es, como sabes, un tipo de imagen con unas aplicaciones muy concretas: suelen ser imágenes de un tamaño reducido y muy concreto (16x16, 32x32 ó 48x48 píxeles es lo más común) que se utilizan por ejemplo como indicadores de acciones en una barra de botones de una aplicación, o aparecen en un listado de una carpeta,etc. Los formatos de este tipo de imágenes pueden ser diversos , pero nosotros utilizaremos siempre el formato PNG (con profundidad de 32 bits) cuando los queramos crear o editar, como si fueran cualquier otra imagen estándar (ya que de hecho, lo son). Así que no necesitaremos más herramientas de las que ya disponemos para las manipular imágenes de nuestro juego: cualquier editor de mapa de bits ó vectorial (que exporte a mapa de bits) servirá perfectamente.

En tu sistema operativo ya vienen instalados multitud de iconos preestablecidos (el icono "carpeta", el icono "documento de texto" , el icono "Mi PC",etc). En Windows suelen llevar la extensión .ICO y situarse en diferentes subcarpetas bajo la carpeta de instalación del sistema. En GNU/Linux normalmente son de extensión PNG y se encuentran bajo la carpeta */usr/share/icons*, subclasificados por tamaños y colecciones. No obstante, la mayoría de iconos -en ambos sistemas- no son archivos separados: no todos los iconos tienen su propio fichero.¿Por qué? Porque los iconos también se pueden encontrar en otro sitio.

Cualquier ejecutable (y también bastantes librerías -DLL en Windows, SO en GNU/Linux-, a parte de incluir el código compilado en binario del propio programa, tiene un apartado donde, dentro del mismo fichero,

almacena uno o más iconos. ¿Y esto por qué? Porque si te has fijado, cada programa tiene un icono distintivo: sería un poco pesado que además del archivo ejecutable (o la librería) el programador que hubiera creado el juego tuviera además que acompañarlo de uno o más iconos separados, con la posibilidad de que se borrara alguno accidentalmente y quedara pues una referencia en el ejecutable a ese archivo inválida. La alternativa es entonces incorporar dentro del propio fichero ejecutable (o librería) los iconos, integrándolos en el código binario como una bacteria fagocitada por una ameba. De esta manera, se tiene sólo el archivo ejecutable con todos los iconos que usará, en su interior. Puedes hacer la prueba de lo que se acaba de comentar. Por ejemplo, en Windows, abre el Explorador de Windows y en el menú "Herramientas" selecciona "Opciones de carpeta". Allí, elige la pestaña "Tipo de fichero" y clica en el botón "Avanzadas". En la ventana que sale, clica en el botón "Cambio de icono…" y allí podrás navegar y elegir cualquier archivo EXE o DLL del sistema. Verás que aparecerán todos los iconos que existen en el interior de éstos. Concretamente, los iconos estándares del escritorio de Windows están en el archivo "shell32.dll", dentro de la carpeta "System32" del sistema.También hay iconos chulos en el archivo "moricons.dll".

Anteriormente se comentó que si queremos que en la barra de títulos (es decir, la barra de color azul que aparece en la parte superior de la ventana)de nuestro juego –si éste no se ejecuta a pantalla completa-) aparezca otro icono diferente que no sea el del dios egipcio naranja, deberemos utilizar la función **set_icon()**, definida en el módulo "mod_wm". Esta función tiene dos parámetros: el primero es la librería FPG que contiene nuestro icono (aunque también podríamos haber cargado perfectamente éste por libre con **load_png()**: en cuyo caso el valor de este primer parámetro sería 0) y el segundo es el código del gráfico dentro del FPG (o bien, el identificador que devuelve **load_png()** si se ha hecho uso de esta función para cargar el icono). El icono puede ser de cualquier tamaño, pero sólo se visualizarán los primeros 32x32 píxeles comenzando por su esquina superior izquierda, así que lo más lógico es diseñar nuestros iconos con ese tamaño.

Recuerda que es necesario llamar a esta función antes que llamar a **set_mode()** , ya que ésta es la función que creará y visualizará dicha ventana. E importante: no debes nunca borrar este gráfico de memoria usando funciones como **unload_fpg()** o **unload_map()** .No es posible cambiar el icono de una ventana una vez ya inicializada, pero se puede volver a llamar a **set_mode()** para activar los cambios.

Por otro lado, la función **set_title()**, definida también en el módulo "mod_wm", se encarga, tal como su nombre indica, a escribir en la barra de títulos el texto especificado como su primer y único parámetro.

Por ejemplo: si tenemos un archivo llamado "iconos.fpg" que representa un FPG que incluirá el icono de nuestro juego,el cual tendrá el código 001, el programa necesario para mostrar el icono deseado en la barra de títulos y un texto concreto sería algo parecido a esto:

```
Import "mod_video";
Import "mod_map";
Import "mod_wm";

Process Main()
Private
          int iconfpg;
End
Begin
          iconfpg=load_fpg("iconos.fpg");
          Set_title("Prueba");
          Set_icon(iconfpg,1);
          Set_mode(640,480,MODE_32BITS,MODE_2XSCALE);
          Loop
                    Frame;
          end
End
```

Después de todo lo explicado, y después de haber probado el código de ejemplo anterior, es posible que hagas algún juego utilizando **set_icon()** y lo quieras distribuir. Tal como comenté en un capítulo anterior, para ello hay que renombrar el intérprete de Bennu con el nombre de tu juego. Pero queda un detalle pendiente y es que el icono de *bgdi* es el círculo mostrando un fénix naranja, pero tu querrás que el icono de tu ejecutable sea el icono que has diseñado tú y que aparece –eso sí- en la barra de título mientras el juego está en marcha. ¿Cómo se pueda cambiar el icono del *bgdi* entonces?

El icono del dios egipcio naranja está integrado dentro del propio fichero que conforma el intérprete de Bennu (*bgdi.exe* en Windows, *bgdi* en GNU/Linux). Por lo tanto, una solución sería "destripar" el código binario de

dicho intérprete, “extirpar” el icono del ave naranja e “implantar” nuestro icono dentro, para que el ejecutable resultante tuviera ahora sí una apariencia totalmente profesional. Este proceso se puede realizar mediante utilidades llamadas editores de recursos.

Un recurso es un icono, o un cursor, o una cadena de texto que está integrada dentro de un ejecutable. Por tanto, con un editor de recursos podríamos modificar, suprimir o añadir iconos, cursores o frases a un ejecutable, sin alterar su correcto funcionamiento (puede ser una manera, por ejemplo, de traducir una aplicación: editando los recursos de cadenas de texto). Nosotros lo podríamos usar para lo que te he comentado: para explorar y localizar dentro del ejecutable el icono propio de *bgdi* y sustituirlo por el nuestro.

Editores de recursos (sólo para Windows) pueden ser:

**ResEdit** (http://www.resedit.net)
**Resource Explorer** (http://www.sharewaresoft.com o,http://www.brothersoft.com, entre otros)
**XN Resource Editor:** http://www.wilsonc.demon.co.uk/d10resourceeditor.htm
**Resource Hakcer:** (http://www.angusj.com/resourcehacker)
**Resource Hunter e Icon Hunter:** http://www.boilsoft.com/rchunter.html e
http://www.boilsoft.com/iconhunter.html -Trial version-
**Restorator:** (http://www.bome.com/Restorator) -Trial version-
**Resource Tuner:** (http://ww.restuner.com) -Trial version-
**Resource Builder:** (http://www.resource-builder.com) -Trial version-

Otra solución, tal vez más sencilla (y válida para GNU/Linux), es obtener el código fuente de Bennu, cambiar el fichero *bgdi.ico* presente allí por el gráfico que deseemos y compilar de nuevo Bennu para obtener la versión modificada del intérprete.

## *Componentes de color en el modo de vídeo de 32 bits: órdenes "Rgba()" y "Rgb()" Orden "Set_text_color()"

A partir de ahora, a menudo necesitaremos especificar un color concreto para seguidamente pintar por ejemplo una región de pantalla con él, o para colorear un texto ó mapa, ó para lo que necesitemos en cada momento. En el modo de 32 bits, la "generación" de un color listo para ser usado se realiza con la función **rgba()**, definida en el módulo "mod_map". En realidad, cada color viene identificado por un número. De hecho, lo que hace la función **rgba()** es devolver un número, un único número resultado de realizar complicados cálculos matemáticos a partir de las cantidades de las componentes R,G,B y Alpha (ver Apéndice B para más información). Y este número entero es el que realmente identifica al color en sí. Pero no se puede saber de antemano qué número corresponde a un determinado color: el resultado del complejo cálculo de mezcla depende de la tarjeta gráfica que se tenga y del modo de vídeo en el que trabajemos (no devolverá **rgba()** el mismo valor para las mismas componentes en modo de 16 bits que en 32, por ejemplo). Es decir, que poniendo el número 25 por ejemplo podríamos encontrarnos que para un ordenador el color nº 25 es parecido al fucsia y en otro ordenador es más parecido al marrón. Por lo tanto, es más seguro siempre utilizar **rgba()**

Esta función tiene cuatro parámetros, que son respectivamente la cantidad de rojo que queremos para nuestro nuevo color, la cantidad de verde, la cantidad de azul, y el nivel de transparencia del color. El mínimo es 0 (en los tres primeros parámetros este valor significa que no hay nada de esa componente en nuestro color y en el último significa que dicho color es totalmente transparente) y el máximo es 255 (en los tres primeros parámetros este valor significa que hay la máxima cantidad posible de esa componente en nuestro color y en el último significa que dicho color es totalmente opaco). Así pues, para generar un negro totalmente opaco tendríamos que escribir algo así como **rgba(0,0,0,255)**, y para generar un blanco semitransparente sería **rgba(255,255,255,130),** por ejemplo. Para generar el rojo más intenso, tendríamos **rgba(255,0,0,255)**; asimismo, para el verde más intenso **rgba(0,255,0)**; y para un azul intenso pero totalmente transparente, con lo que no se vería en realidad ningún color, **rgb(0,0,255,0);**, y a partir de aquí, con todas las combinaciones que se nos ocurran podremos crear los colores que queramos con el nivel de opacidad que deseemos.

Existe otra función, **rgb(),** definida en el mismo módulo, que es equivalente a **rgba()** excepto en el sentido de que no establece ninguna información sobre el canal alfa: sólo tiene tres parámetros que son los

correspondientes a los canales R,G y B. De hecho, se presupone que el valor de la opacidad utilizando esta función es el máximo (255), es decir, totalmente opaco. Por lo tanto, una línea como *rgba(34,53,167,255);* sería equivalente a *rgb(34,53,167);*. Esta función es la que debería usar preferiblemente en el modo de vídeo de 16 bits.

También existe una función inversa a **rgba()** llamada **get_rgba()** -definida también en "mod_map"-, que a partir de un número entero que identifica a un color, obtiene la cantidad correspondiente de su componente roja,verde, azul y alpha. Esta función es un poco más avanzada y la veremos en próximos capítulos. También existe la equivalente sin alpha **get_rgb()**.

A continuación presento un código de ejemplo de uso de la función **rgba()**. Aprovecho la ocasión para introducir también en este ejemplo una función que quedó pendiente del capítulo anterior cuando hablábamos de los textos en pantalla: **set_text_color()** -definida en el módulo "mod_text"-. Esta función colorea con el color que nosotros establezcamos exactamente mediante **rgba()** todos los textos que se escriban con **write()** ó **write_var()** posteriormente a su ejecución , siempre que no se utilice una fuente de letra que ya tenga colores intrínsecos (como es nuestro caso hasta ahora, ya que utilizamos la fuente del sistema). Así:

```
Import "mod_text";
Import "mod_video";
Import "mod_map";
Import "mod_key";

Process Main()
Private
	int color;
end
begin
	set_mode(320,200,32);
	loop
		write_var(0,100,50,4,color);
		if (key(_r))
			delete_text(0);
			color=rgba(255,0,0,255);
			set_text_color(color);
			write_var(0,100,50,4,color);
			write(0,100,100,4,"Hola");
		end
		if (key(_g))
			delete_text(0); color=rgba(0,255,0,255); set_text_color(color);
			write_var(0,100,50,4,color); write(0,100,100,4,"Hola");
		end
		if (key(_b))
			delete_text(0); color=rgba(0,0,255,255); set_text_color(color);
			write_var(0,100,50,4,color); write(0,100,100,4,"Hola");
		end
		if (key(_a))
			delete_text(0); color=rgba(64,243,28,0); set_text_color(color);
			write_var(0,100,50,4,color); write(0,100,100,4,"Hola");
		end
		if (key(_y))
			delete_text(0); color=rgba(255,255,0,0); set_text_color(color);
			write_var(0,100,50,4,color); write(0,100,100,4,"Hola");
		end
		if (key(_p))
			delete_text(0); color=rgba(255,0,255,0); set_text_color(color);
			write_var(0,100,50,4,color); write(0,100,100,4,"Hola");
		end
		if (key(_o))
			delete_text(0); color=rgba(0,255,255,0); set_text_color(color);
			write_var(0,100,50,4,color); write(0,100,100,4,"Hola");
		end
		if (key(_esc))
			break;
		end
		frame;
	end
end
```

Lo primero que hacemos es mostrar el valor de la variable *color*, que será la que contendrá el valor número devuelto por **rgba()** correspondiente al color elegido en un momento dado para pintar posteriormente los textos con **set_text_color()**. Fijarse como, si no se pulsa ninguna tecla, a esta variable nunca se le asignará ningún valor, por lo que siempre permanecerá con su valor por defecto inicial, que es 0. En el momento que se pulse la tecla "r", *color* pasa a valer el valor numérico correspondiente al color rojo puro, se establece que los textos escritos a partir de entonces tendrán ese color y se escribe a continuación precisamente con ese color un texto cualquiera y el nuevo valor de *color*, que será un número imposible de determinar con antelación porque ya hemos dicho que es dependiente del modo de vídeo (es decir, en otra profundidad de color la **rgba()** devuelve otro valor diferente para las mismas componentes) y de la tarjeta gráfica. Si pulsamos la tecla "g", se borran todos los posibles textos anteriores (para empezar de cero otra vez), se vuelve a asignar a *color* otro valor de color, se vuelve a establecer que ese color será el de los próximos textos, y se vuelve a escribir uno cualquier y el nuevo valor de *color*, correspondiente al verde. Y así, si pulsamos "b" veremos el texto en azul, si pulsamos "y" veremos amarillo, con "p" será púrpura y con "o" un azul eléctrico.

Fíjate, no obstante, que si pulsamos "a" no hace lo que se supone que debería hacer. En teoría, deberían de dejarse de ver los textos porque el color que hemos establecido es uno cualquiera (un verde) pero con la componente alpha a 0, por lo que en realidad ese color es transparente. Pero sin embargo, el texto se sigue viendo con ese color verde, así que parece que la información de transparencia no la toma en cuenta. No te preocupes, eso es normal. Esto es debido a que los textos que no se escriben con una fuente de letra de 32 bits (porque, ojo, las fuentes de letra también tienen su profundidad de color), lógicamente no utilizan la información de transparencia. Y la fuente del sistema, que es la que estamos utilizando siempre (recuerda, eso es lo especificamos poniendo 0 como valor del primer parámetro del **write()** ) es una fuente de letra de 1 bit de profundidad de color. Eso implica que cada letra puede tener sólo un color posible estableciendo su bit respectivo a "1", y ese color normalmente será blanco o si se establece con **set_text_color()**, el que se haya dicho. Pero nada de transparencias. Sobre el tema de las fuentes hablaremos en capítulos posteriores.

Finalmente, comentar que las dos líneas siguientes del código anterior:

```
color=rgba(255,0,0,255);
set_text_color(color);
```

se podrían haber condensado en una sola línea de la siguiente manera:

```
set_text_color(rgba(255,0,0,255));
```

Es decir, en vez de haber tenido que utilizar una variable, podríamos haber incluido directamente la función **rgba()** como el parámetro de **set_text_color()**, de manera que el valor devuelto por **rgba()** fuese recogido automáticamente por **set_text_color()** sin pasos intermedios. Ambas formas son igual de correctas: es una cuestión de preferencia personal elegir entre una u otra.

También existe otra función, **GET_TEXT_COLOR()**, la cual no tiene ningún parámetro y lo que devuelve es un número que corresponde el valor numérico del color con el que en este momento se están escribiendo los textos. Este valor numérico, vuelvo a insistir, dependerá del modo de vídeo y de la tarjeta gráfica concreta que se esté utilizando, por lo que puede variar de ordenador a ordenador.

## *Transparencias en el modo de vídeo de 32 bits

Es importante saber cómo funcionan las transparencias en Bennu, porque de ello depende por ejemplo el sistema de colisiones (“choques”) entre los distintos gráficos de nuestro juego, entre otras muchas cosas. Si no tomas las precauciones que pasaré a indicar a continuación, puede que algunos píxeles que aparentemente en tu imagen se ven transparentes no lo sean en realidad a la hora de causar colisiones. Esto es debido al efecto llamado “cristal”: un cristal no se ve pero colisiona: pasa lo mismo con los píxeles transparentes.

Como sabemos, el píxel visualmente transparente se produce al llevar su componente alfa al valor 0, pero recordemos que sigue teniendo información en sus componentes RGB, así que tanto el verde con alfa 0 como el amarillo con alfa 0,etc no se verán en la imagen. Pero son colores distintos: uno es el verde transparente total y otro es el amarillo transparente total aunque no se vean ninguno de los dos, y Bennu sabrá que por detrás hay o un verde o un

amarillo que no se ven y hará que se produzcan colisiones. ¿Entonces, cuál es el color transparente "de verdad"? Ahora viene lo importante: Bennu ha sido programado para que el único color transparente total que no dé colisiones sea el negro transparente total, o sea las componentes RGB negras (R=0, G=0, B=0) y alfa 0. Es decir, el color transparente total en Bennu es aquél que su representación interna en 32 bits es un 0 absoluto: R = 0, G = 0, B = 0, Alpha = 0.

Hay que tener en cuenta este detalle a la hora de dibujar los sprites, ya que no todo lo que se vea transparente tiene obligatoriamente que serlo. Para asegurarse, lo que hay que hacer con cualquier editor de imágenes es eliminar toda la información previa sobre transparencias que pudiera tener la imagen (por si acaso), y pintar las zonas que queremos que sean transparentes con el color negro absoluto, para finalmente seleccionar dichas zonas y convertir el color negro en alfa. Siguiendo este procedimiento, nos aseguramos que tantos las componentes de color RGB y el canal alpha de las zonas que queremos que sean transparentes valgan 0.

Un truco para ver qué colores hay detrás del canal alpha de una imagen y así revelar qué regiones transparentes van a producir colisiones es por ejemplo abrir la imagen en el programa Xnview y en el menú "View" desactivar la opción "Use alfa channel" Así se verá la imagen sin el canal alfa y se verán las regiones transparentes en el color de su componente RGB.

## *Sobre el uso de paletas de colores en el modo de vídeo de 8 bits

Anteriormente hemos comentado de pasada la existencia de las llamadas paletas de colores, (en el Apéndice B se explica mejor lo que son). Incluso hay un formato de archivo propio de Bennu (los archivos .PAL) que se dedica a guardarlas en disco. Pero en realidad, estas paletas sólo tiene sentido utilizarlas cuando trabajamos con gráficos de 8 bits, y como en este manual se trabajará siempre con gráficos de 32 bits, no profundizaremos en este tema. Por ejemplo, Bennu aporta muchas funciones para trabajar con paletas como CONVERT_PALETTE(), ROLL_PALETTE(), FIND_COLOR(), GET_COLORS(), SET_COLORS(), LOAD_PAL(), SAVE_PAL(),etc que no se explicarán. No obstante, merece la pena introducir el posible uso que podrían tener las paletas en el caso que se hicieran servir.

Lo primero que hay que saber es que sólo se puede tener una paleta cargada a la vez. Cuando Bennu carga una imagen de 8 bits desde archivo, guardará su paleta de colores que lleva incorporada y la usará para el resto de imágenes a cargar. Esto obliga a que todos los sprites usados en modo de 8 bits deben de realizarse siempre con la misma paleta de colores, ya que si no se producirían efectos de cambios de color inesperados.

Debido a que una paleta de 8 bits tan sólo puede almacenar 256 colores en pantalla, no se podrán crear imágenes muy realistas (degradados de color suaves, etc). Como mal menor, se suelen elegir para incorporar a la paleta los 256 colores más presentes en la representación de una imagen. Por eso, para mostrar una imagen de una selva, se usarían muchos tonos de verdes, algunos naranjas o rojos, y menos azules. O para representar un cielo,la mayoría de los colores serian azules, en sus tonalidades desde el negro hasta el blanco, etc. Cuando se trabaja en modo 16 bits ó 32 bits no se necesitan paletas porque hay muchísimos más colores disponibles (por ejemplo, en 16 bits: $2^{16}$ = 65536 colores). Entonces ya no tenemos que elegir colores :se pueden utilizar tantos que todas las imágenes tendrán buena calidad. En esos modos se definen los colores estableciendo la intensidad deseada de cada componente de color R,G,B -y Alpha en 32 bits-.

Si queremos que nuestros juegos vayan rápidos independientemente del ordenador en el que vayan a ejecutarse por muy antiguo que sea, o si tenemos pensado que nuestro juego funcione sobre algún dispositivo móvil o consola portátil (con menos recursos hardware), entonces tendría sentido plantearnos usar paletas de 8 bits (256 colores). Ya hemos dicho que al hacerlo tendremos que elegir bien estos 256 colores para sacarles el mayor partido posible: así que tendremos que construir nuestra paleta de 256 colores de acuerdo a los que vayamos a utilizar , mediante programas de edición de gráficos como Gimp y con la ayuda de varios comandos de Bennu de manipulación de paletas que podrán ser usados durante la ejecución de un juego, bien para cambiarlas por completo, bien para cambiar colores individualmente, etc. De todas maneras, piensa que nada te impide usar varias paletas por turnos (por ejemplo, en un juego de plataformas con una 1ª fase de hielo -predominarán los blancos y los azules- y una 2ª de fuego -tonos rojos-; al pasar de la fase 1 a la 2 se descargaría de memoria (o simplemente se dejaría de utilizar) los gráficos (contenidos en un fpg, por ejemplo) de la 1ª y se cargaría los gráficos contenidos en otro fpg (y por consiguiente las paletas) de la 2ª.

Hemos dicho que podemos usar un editor gráfico para crear una paleta. Pero, ¿eso cómo se hace? En

general no será necesario crear explícitamente ninguna paleta, porque cuando se crea una imagen de 8 bits, ésta ya incorpora dentro su propia paleta. También es verdad que la mayoría de programas gráficos dan la opción, a partir de una determinada imagen de 8 bits, de guardar las paletas por separado, para así al crear una nueva imagen se le pueda asociar alguna paleta guardada previamente.

El uso de 256 colores tiene otras ventajas, y no sólo el ahorro de memoria o mayor velocidad de ejecución: te permite tener control total sobre los colores que usas, pues manejar 256 colores es mas sencillo que los 65000 y pico del modo 16 bits o los millones del 32 bits; sobre todo porque cada "casilla" (es decir, cada color de la paleta) lo puedes modificar para conseguir efectos de brillos, cambios de color... e incluso, con conocimientos de los efectos del color, modificando una paleta entera puedes usar una misma imagen para representar el día, la noche y todas las tonalidades intermedias, cosa que en los otros modos puede ser algo tedioso porque se tendría que modificar todos los gráficos, en lugar de 256 "casillas".

Por último, indicar que en el modo de vídeo de 8 bits de Bennu se pueden usar transparencias aunque la imagen propiamente no lo sea. Para Bennu, sea cual sea el color que esté presente en el índice número 0 de la paleta cargada (el primer color), ése será el color transparente. Así que para diseñar las imágenes PNG de 8 bits necesarias habrá que tener en cuenta que el color que queramos usar como transparente para nuestros sprites se debe de encontrar en la primera posición de la paleta de colores a utilizar.

## *Trabajar con gráficos generados dinámicamente en memoria

En determinadas ocasiones, no nos hará falta tener gráficos creados independientemente de nuestro juego ni usar contenedores FPG externos. Bennu es capaz de generar y/o manipular gráficos/sprites directamente desde código: es lo que se dice generación dinámica en memoria. Además, estos gráficos generados "a partir de la nada" ,así como también los gráficos cargados de disco que podamos haber modificado en la RAM (porque ya sabemos que para Bennu todos los gráficos -"mapas"- en la memoria son iguales) los podemos grabar en un fichero, de manera que Bennu nos puede servir también como editor de dibujos rudimentario.

A continuación presento las funciones más importantes de generación y manipulación dinámica de imágenes.

**NEW_MAP ( ANCHO, ALTO, PROFUNDIDAD )**

Esta función, definida en el módulo "mod_map", crea un nuevo gráfico en memoria, con el tamaño y número de bits por pixel de color especificados. El gráfico se crea dentro de la librería 0 -es decir, la librería del sistema, fuera de cualquier FPG-, y se le asigna un número identificador cualquiera disponible que no haya sido usado por ningún otro gráfico anterior.

El nuevo gráfico se crea con todos los píxeles transparentes. Por lo tanto, al crearlo no se verá nada en pantalla. El objetivo de esta función es reservar el espacio en memoria necesario para alojar el gráfico y para que, posteriormente, mediante el uso de otras funciones, se pueda pintar en pantalla

Se puede elegir que el gráfico tenga una profundidad (además de las ya conocidas 8, 16 y 32) de 1 bit. En este caso, no hay información de color: los píxeles a 0 representan píxeles transparentes y los bits a 1 serán dibujados usando el color escogido mediante la función **drawing_color()**, la cual explicaremos hacia el final de este capítulo.

PARÁMETROS:

INT ANCHO : Ancho en pixels del nuevo gráfico
INT ALTO : Alto en pixels del nuevo gráfico
INT PROFUNDIDAD : Profundidad de color (en bits por pixel: 1, 8 , 16 ó 32)

**MAP_PUT_PIXEL ( LIBRERÍA, GRÁFICO, X, Y, COLOR ) | PUT_PIXEL(X,Y,COLOR)**

Esta función, definida en el módulo "mod_draw", permite alterar el color de un pixel determinado dentro de un gráfico especificado. Las coordenadas dadas crecen hacia abajo y a la derecha, y el punto (0, 0) representa la esquina superior izquierda del gráfico.

En el caso de gráficos de 256 colores (8 bits), el valor de color debe ser un número de 0 a 255. En el caso de gráficos de 16 y 32 bits, el valor es una codificación numérica de las componentes del color que varía en función de la tarjeta de vídeo y el modo gráfico. Lo normal es usar la función **rgb()** ó **rgba()** para obtener la codificación de un color concreto, o bien usar un color obtenido por una función como **map_get_pixel()** -la estudiaremos enseguida-.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG, ó 0 si el gráfico ha sido cargado individualmente
INT GRÁFICO : Número de gráfico dentro del FPG, ó su identificador si ha sido cargado individual.
INT X : Coordenada horizontal
INT Y : Coordenada vertical
INT COLOR : Color a dibujar

Si se quiere realizar la misma función pero no sobre un gráfico cualquiera sino específicamente sobre el fondo de pantalla, se puede hacer uso de la librería 0 gráfico 0 dentro de **map_put_pixel()** -es decir,escribiendo *map_put_pixel(0,0,x,y,color);*-, o bien se puede utilizar por comodidad la función **PUT_PIXEL(X,Y,COLOR)**, cuyos tres parámetros tienen idéntico significado que los últimos tres de **map_put_pixel()**, y donde se sobreentiende que el gráfico sobre el que se pintará es el del fondo.

Con estas dos funciones ya podemos generar nuestro primer gráfico "al vuelo" dentro de Bennu. Un ejemplo podría ser éste:

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";

Process Main()
private
   int a;
   int b;
   int idgraf;
end
begin
set_mode(640,480,32);
set_fps(60,1);
idgraf=new_map(30,30,32);
for (a=0;a<=30;a++)
      for(b=0;b<=30;b++)
            map_put_pixel(0,idgraf,a,b,rgba(25,78,192,156));
      end
end
put_screen(0,idgraf);
loop
      if(key(_esc)) break; end
      frame;
end
end
```

Lo que hacemos en este ejemplo es, después de especificar el modo de vídeo y los fps, crear una imagen en la memoria de dimensiones 30x30 de 32 bits con **new_map()**. Esta imagen a partir de entonces estará cargada (como si la hubiéramos cargado con **load_png()**, por ejemplo) y lista para ser referenciada usando para ello su identificador, devuelto por **new_map()** y guardado en *idgraf*. Pero no hemos definido todavía qué gráfico veremos en ella: es una imagen "sin imagen". Para "pintar" la imagen utilizamos **map_put_pixel()** (aunque no es el único método para ello: ya lo veremos). En concreto, con esta función pintamos con un color especificado -en nuestro ejemplo, un azul semitransparente- un píxel especificado -dado por sus coordenadas x e y- de un gráfico especificado -en nuestro ejemplo, el gráfico que acabamos de crear cuyo identificador está guardado en *idgraf* (recordemos que al no pertenecer la imagen a ninguna librería FPG, el primer parámetro de **map_put_pixel()** ha de valer 0)-.

Como **map_put_pixel()** sólo pinta un pixel cada vez que se ejecuta, hay que ejecutarla muchas veces para pintar todos los píxeles que ocupan el área de 30x30 de la imagen, y además especificando cada vez las coordenadas de un píxel diferente. Para recorrer, pues, todos los píxeles que forman el cuadrado, llamamos a esta función dentro de un par de bucles anidados: de esta manera el tercer y cuarto parámetro de la función, que indican qué

pixel hay que colorear, valdrán desde (0,0) -esquina superior izquierda del gráfico- hasta (30,30) -esquina inferior derecha-.

Sin embargo, no basta con pintar los píxeles con **map_put_pixel()** para que se muestre en pantalla la imagen. Los píxeles se han pintado,sí, pero en memoria otra vez. Ahora tenemos la imagen "hecha", pero no hemos dicho en ningún sitio que se muestre en ningún lugar concreto de la pantalla. Es decir, existe, pero no la hemos plasmado en ningún sitio. De hecho, de momento la única manera que hemos visto (y conocemos) de plasmar imágenes en la pantalla es poniéndolas como imágenes de fondo con **put_screen()**, así que en este ejemplo es eso lo que hemos hecho: ponerla como imagen de fondo y por lo tanto, poderla visualizar al fin. Fíjate que **put_screen()** la hemos escrito después de haber creado la imagen y sobretodo, después de haberla pintado, ya que si, por ejemplo, hubiéramos puesto **put_screen()** antes de los bucles, no hubiéramos visto nada en pantalla, porque en el momento de poner la imagen de fondo, ésta todavía no habría estado pintada, y aunque lo hubiera estado después, el fondo no se habría visto afectado: hubiera habido que volverlo a actualizar.

A continuación, presento una variación del ejemplo anterior:

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";

Process Main()
private
   int a;
   int b;
   int idgraf;
end
begin
set_mode(640,480,32);
set_fps(60,1);
idgraf=new_map(30,30,32);
for (a=0;a<=30;a++)
      for(b=0;b<=30;b++)
            map_put_pixel(0,idgraf,a,b,rgba(25,78,192,156));
            put_screen(0,idgraf);
            frame;
      end
end
loop
      if(key(_esc)) break; end
      frame;
end
end
```

La única diferencia con el ejemplo anterior ha sido quitar **put_screen()** de donde estaba e incluirlo dentro de los bucles anidados, además de también incluir allí una orden frame. ¿Qué es lo que ocurre ahora cuando ejecutas el programa? Verás que el cuadrado aparece poquito a poquito, pintándose, hasta llegar a quedar pintado completamente. Esto ocurre porque en cada iteración estamos cambiando la imagen de fondo por otra que es la misma que la anterior pero con un píxel más pintado. Es decir, estamos viendo "en tiempo real" el proceso de pintado de la imagen en memoria, plasmándolo inmediatamente en pantalla también. Para lograr este efecto, la presencia de la orden frame es fundamental, para poder visualizar el resultado de **put_screen()** a cada iteración: si no lo pusiéramos (pruébalo), la imagen de fondo iría cambiando pero no lo podríamos visualizar porque no llegaría ningún frame hasta el bucle final del loop, momento en el cual la imagen de fondo que tendríamos sería ya la imagen totalmente acabada. Así que sin frame volveríamos a ver tan sólo el cuadrado completamente acabado, a pesar de haber ido cambiando las imágenes de fondo unas decenas de veces.

Otro ejemplo, parecido al anterior, ofrece un efecto curioso:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_screen";
Import "mod_rand";
```

```
Import "mod_draw";

Process Main()
private
    int idgraf;
    int numcolor;
end
begin
    set_mode(640,480,32);
    set_fps(100,1);
    idgraf=new_map(640,480,32);
    repeat
         numcolor=rgba(rand(0,255),rand(0,255),rand(0,255),rand(0,255));
         map_put_pixel(0,idgraf,rand(0,639),rand(0,479),numcolor);
         put_screen(0,idgraf);
         frame;
    until(key(_esc))
end
```

Creo que viendo el resultado es fácil de entender lo que hace este código.

Otras funciones interesantes son:

**MAP_CLEAR ( LIBRERÍA, GRÁFICO, COLOR )**

Esta función,definida en el módulo "mod_map", borra el contenido anterior de un determinado gráfico en memoria, sin descargarlo, y pinta seguidamente todos sus píxeles con el color indicado. Es otra manera de decir que rellena de un color el gráfico completo, por lo que utilizar esta función es una manera de pintar rápidamente un gráfico creado anteriormente con **new_map()**.

Como siempre, el significado del parámetro color depende de la profundidad de color del gráfico. En el modo de 32 bits, usaremos siempre la función **rgba()** para definirlo.

Recuerda que, en cualquier caso, el color con todas las componentes y el alfa igual a 0 representa un píxel transparente. Y recuerda también que el gráfico perteneciente a la librería 0 (la del sistema) cuyo código identificador es 0 corresponde a la imagen de fondo de pantalla.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico dentro de la librería
INT COLOR : Número de color

Un ejemplo donde se puede ver el cambio repetido de color que sufre un gráfico gracias a repetidas llamadas a **map_clear()** es:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_screen";
Import "mod_rand";

Process Main()
private
  int idgraf;
end
begin
  set_mode(640,480,32);
  set_fps(1,1); //Reducimos la velocidad de los fps para que el ejemplo sea visible
  idgraf=new_map(300,200,32);
  repeat
      map_clear(0,idgraf,rgba(rand(0,255),rand(0,255),rand(0,255),rand(0,255)));
      put_screen(0,idgraf);
      frame;
  until(key(_esc))
```

```
end
```

| MAP_GET_PIXEL ( LIBRERÍA, GRÁFICO, X, Y ) | GET_PIXEL(X,Y) |
|---|---|

Esta función, definida en el módulo "mod_draw", recupera el color de un píxel determinado dentro de un gráfico determinado. Las coordenadas dadas crecen hacia abajo y a la derecha, y el punto (0, 0) representa la esquina superior izquierda del gráfico.

En el caso de gráficos de 256 colores (8 bits), el valor devuelto por esta función es un color perteneciente a la paleta cargada en ese momento (un valor entre 0 y 255) . En el caso de gráficos de 16 bits y 32 bits, el valor devuelto es una codificación de las componentes del color que depende de la tarjeta de vídeo y el modo gráfico. Puede usarse **get_rbga()** para obtener a partir de esa codificación, los valores por separado de las componentes del color (que nos será más útil).

Esta función será de vital importancia cuando se trate el tema de los mapas de durezas (visto en el capítulo-tutorial de RPG). En su momento se explicará pormenorizadamente su funcionalidad y diferentes usos.

Si se quiere realizar la misma función pero específicamente sobre el fondo de pantalla, o bien se hace uso de la librería 0 gráfico 0 dentro de **map_get_pixel()** -es decir, se escribe *map_get_pixel(0,0,x,y)*-, o bien se puede utilizar por comodidad la función **GET_PIXEL(X,Y)**, cuyos dos parámetros tienen idéntico significado que los últimos dos de **map_get_pixel()**, y donde se sobreentiende que el gráfico sobre el que se pintará es el del fondo.

PARÁMETROS:
- INT LIBRERÍA : Número de librería FPG
- INT GRÁFICO : Número de gráfico dentro de la librería
- INT X : Coordenada horizontal
- INT Y : Coordenada vertical

Como ejemplo introductorio a la funcionalidad de este comando, aquí va el siguiente código, suficientemente claro:

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";
Import "mod_text";
Import "mod_rand";

Process Main()
private
  int a,b;
  int fondo;
  int color;
  int barra;
end
begin
  set_mode(640,480,32);
/*Las líneas siguientes no son más que bucles para generar un fondo formado por
diferentes columnas de colores, a modo de "carta de ajuste". Como se puede ver, las
columnas son de 100 píxeles de ancho y cada una se pinta de un color*/
  fondo=new_map(640,480,32);
  for(a=0;a<100;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(255,0,0));
        end
  end
  for(a=100;a<200;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(0,255,0));
        end
  end
```

```
  for(a=200;a<300;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(0,0,255));
        end
  end
  for(a=300;a<400;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(255,255,0));
        end
  end
  for(a=400;a<500;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(255,0,255));
        end
  end
  for(a=500;a<600;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(0,255,255));
        end
  end
  for(a=600;a<640;a++)
        for(b=0;b<480;b++)
           map_put_pixel(0,fondo,a,b,rgb(255,255,255));
        end
  end
  put_screen(0,fondo);

  write(0,10,10,3,"Color de la pantalla: ");
  write_var(0,200,10,3,color);
  write(0,10,20,3,"Barra nº: ");
  write_var(0,200,20,3, barra);
  repeat
      if(key(_r)) //Elegimos un punto aleatorio de la primera barra
           a=rand(0,99);
           b=rand(0,479);
           color=map_get_pixel(0,fondo,a,b);
           barra=1;
      end
      if(key(_g)) //Lo mismo para un punto de la segunda barra
           a=rand(100,199); b=rand(0,479); color=map_get_pixel(0,fondo,a,b); barra=2;
      end
      if(key(_b))
           a=rand(200,299); b=rand(0,479); color=map_get_pixel(0,fondo,a,b); barra=3;
      end
      if(key(_y))
           a=rand(300,399); b=rand(0,479); color=map_get_pixel(0,fondo,a,b); barra=4;
      end
      if(key(_p))
           a=rand(400,499); b=rand(0,479); color=map_get_pixel(0,fondo,a,b); barra=5;
      end
      if(key(_o))
           a=rand(500,599); b=rand(0,479); color=map_get_pixel(0,fondo,a,b); barra=6;
      end
      if(key(_w))
           a=rand(600,639); b=rand(0,479); color=map_get_pixel(0,fondo,a,b); barra=7;
      end
      frame;
  until(key(_esc))
end
```

Un ejemplo curioso donde se emplean **map_put_pixel()** y **map_get_pixel()** es el siguiente. Primero se visualiza como fondo de pantalla una imagen con código 001 perteneciente a un FPG llamado "graficos.fpg". A continuación, se realiza un barrido horizontal donde se aplica un efecto de oscurecimiento. Una vez acabado éste, se realiza otro barrido horizontal donde se aplica un efecto de abrillantamiento.

Par lograr el oscurecimiento, lo único que se hace es convertir el color de todos aquellos píxeles de la imagen que tengan un cantidad de rojo,verde y azul menor de 15, al color negro más absoluto. Con esto, lo que estamos

haciendo es oscurecer los píxeles que eran más oscuros ya de entrada, dejando inalterables los demás. Se puede jugar con el límite que queramos para incluir o excluir píxeles del oscurecimiento: a un mínimo más elevado, abrá más zonas de la imagen que se conviertan en negro. También se puede jugar con la condición del if: si en vez de haber un "<" hay un ">", estaremos logrando el efecto contrario, también muy interesante.

Para lograr el abrillantamiento, se parte de la misma idea: se van recorriendo los píxeles de la imagen y si el color de éstos cumple una condición límite, su color se verá incrementado con el brillo que queremos definir (se puede variar su valor para comprobar el efecto que causa).

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";

Process Main()
private
  int c;
  int min=75,
  int bri=50;
  int tt;
  int idfpg;
end
begin
set_mode(640,480,32);
idfpg=load_fpg("graficos.fpg");
set_fps(100,0);
put_screen(idfpg,1);

//Primer efecto: oscurecimiento
For (x=0;x<640;x++)
          For (y=0;y<480;y++)
                    c=map_get_pixel(0,0,x,y);
                    if (c<rgb(min,min,min))  c=rgb(0,0,0); end
                    map_put_pixel(0,0,x,y,c);
          end
/*La siguiente línea es para visualizar el efecto más velozmente: sólo se ejecuta el
frame cada cinco iteraciones*/
          tt++;if(tt>5) tt=0;frame;end
end

//Segundo efecto: abrillantamiento
For (x=0;x<640;x++)
          For (y=0;y<480;y++)
             c=map_get_pixel(0,0,x,y);
             if (c<rgb(255-bri,255-bri,255-bri)) c=c+rgb(bri,bri,bri); end
             map_put_pixel(0,0,x,y,c);
          end
          tt++;if(tt>5) tt=0;frame;end
end
/*Si quisiéramos grabar en un fichero la imagen resultante de ambos efectos, podríamos
escribir en este lugar la línea: Save_png(0,0,"Hola.png"); La orden save_png se tratará
más adelante*/
repeat
 frame;
until (key(_esc))
end
```

Otro ejemplo de oscurecimiento, similar en la filosofía al anterior pero bastante más complejo en su código, es el siguiente. Necesitarás otra vez un FPG llamado "graficos.fpg" con un gráfico de identificador 001. Si ejecutas el ejemplo,primero verás la imagen tal cual; después de pulsar la tecla SPACE no verás nada (habrás quitado esta imagen del fondo) para entonces volver a pulsar la tecla SPACE y volver a ver la misma imagen de antes, pero con una zona rectangular en su interior donde aparecerá más oscurecida.El tamaño de esta porción de imagen oscurecida lo podrás cambiar (sólo) en el código fuente, el cual, de hecho, es de un nivel más avanzado del que estamos en este momento, y aparecen elementos (como la creación de funciones propias, o el uso de la función **get_rgb()** ) que no hemos visto todavía. No obstante, he considerado que este interesante ejemplo es lo suficientemente explícito y bien

documentado para que su funcionamiento general no sea difícil de entender. Además, copiando y pegando la función "oscurecer()" en cualquier otro código, la podrás reutilizar para poder oscurecer las imágenes que quieras.

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";

Process Main()
private
  int idfpg;
end
begin
set_mode(640,480,32);
idfpg=load_fpg("graficos.fpg");
//Primero pongo la imagen original tal cual
put_screen(idfpg,1);
/*Mientras no se pulse la tecla espacio no se hace nada; es decir, se visualiza la imagen
de fondo y ya está*/
while(not key(_space)) frame; end
//Una vez que he pulsado la tecla SPACE, mientras la tenga pulsada tampoco hago nada
while(key(_space)) frame; end
//Una vez que he soltado la tecla SPACE, borro el fondo de pantalla, quedándose negra
clear_screen();
while(not key(_space)) frame; end
while(key(_space)) frame; end
/*Después de haber pulsado otra vez SPACE, aplico la función "oscurecer" a la misma
imagen de antes*/
oscurecer(idfpg,1,30,30,300,300);
/*Y la vuelvo a mostrar como fondo de pantalla, observando las diferencias con el fondo
anterior (básicamente, la presencia de un cuadrado más oscuro)*/
put_screen(idfpg,1);
repeat
 frame;
until (key(_esc))
end

/*Esta función tiene 6 parámetros: el primero es el fpg donde se encuentra la imagen,
(cuyo identificador es el valor del segundo parámetro), que será oscurecida. No obstante,
no tiene porqué oscurecerse toda la imagen, sino sólo una porción rectangular de esta
indicada por los cuatro parámetros siguientes:el tercero es la coordenada X de la esquina
superior izquierda de esta zona oscura, el cuarto es la coordenada Y de ese punto, el
quinto es la coordenada X de la esquina inferior derecha de esa zona oscura y el sexto es
la coordenada Y de ese punto. Hay que tener en cuenta que estas coordenadas tienen como
origen la esquina superior izquierda de la imagen, no de la pantalla.*/

function oscurecer(int fpg,int imagen,int x1,int y1,int x2,int y2)
private
          int i,j,color;
          byte r,g,b;
end
BEGIN
//Recorro la zona a oscurecer a lo largo y a lo ancho
FOR(i=x1; i<=x2; i++)
          FOR(j=y1; j<=y2; j++)
                    //Obtengo el color original de cada pixel dentro de esta zona
                              color=map_get_pixel(fpg,imagen,i,j);
/*Obtengo, a partir del color obtenido, sus componentes RGB individuales. La función
get_rgb no la hemos visto todavía (se verá en profundidad más adelante en este capítulo),
pero básicamente sirve para esto: a partir de un código único de color dado por
map_get_pixel en este caso, obtenemos los valores independientes de las componentes R,G y
B en sendas variables (los tres últimos parámetros). Atención con poner el & de cada uno
de los nombres de estas variables.*/
                              get_rgb (color, &r, &g, &b);
/*Repinto el mismo pixel del cual había obtenido el color original con un nuevo color
generado a partir de multiplicar por 0.6 (y por tanto, reducir) cada uno de los valores
de las tres componentes del color original. Con esto se logra el efecto de
```

```
oscurecimiento. Si multiplicamos por un decimal menor (0.5,0.4,etc), obtendremos cada vez
valores de las componentes más pequeños y por tanto tenderemos cada vez más a 0 -es
decir, más negro-. Lo puedes probar.¿Qué pasaría si este factor fuera más grande de 1?
Podríamos incluso modificar la función para que admitiera un nuevo parámetro que fuera
precisamente el valor de este factor, para tener una única función que permitiera generar
diferentes tipos de oscurecimientos. Un detalle importante: el hecho de utilizar
map_put_pixel implica que la imagen se ha modificado en memoria, pero de momento no es
visible: necesitamos otra orden que haga que esta (nueva) imagen se vea por pantalla: por
eso en el código principal, después de haber llamado a "oscurecer", volvemos a llamar a
put_screen.*/
                                map_put_pixel(fpg,imagen,i,j, rgb(r*0.6, g*0.6, b*0.6) );
          END
END
END
```

**WRITE_IN_MAP ( FUENTE, "TEXTO", ALINEACIÓN )**

Esta función, definida en el módulo "mod_text", crea un nuevo gráfico en memoria, de la misma manera que la función **new_map()** , pero a partir de un texto. Devuelve un identificador único para el nuevo gráfico creado, de forma similar a **new_map()**, pero el gráfico contendrá el texto especificado como parámetro, en el tipo de letra indicado, y tendrá el tamaño justo para contener todas las letras sin desperdiciar espacio. Formará parte de la librería del sistema, de código 0 y tendrá la misma profundidad de color que el tipo de letra, y píxeles transparentes al igual que los propios caracteres de la fuente.

El parámetro de alineación indica el valor que será tomado por el centro del nuevo gráfico (punto de control cero), y puede ser uno de los siguientes valores:

- 0: el punto ocupará la esquina superior izquierda del gráfico
- 1: el punto ocupará el centro de la primera fila
- 2: el punto ocupará la esquina superior derecha
- 3: el punto ocupará el centro de la primera columna
- 4: el punto ocupará el centro exacto del gráfico
- 5: el punto ocupará el centro de la última columna
- 6: el punto ocupará la esquina inferior izquierda
- 7: el punto ocupará el centro de la fila inferior
- 8: el punto ocupará la esquina inferior derecha

Si no se desea usar el gráfico por más tiempo, es recomendable liberar la memoria ocupada por el mismo mediante la función **unload_map()**

La mayor utilidad de esta función consiste en poder dibujar textos con la misma variedad de efectos que los gráficos normales (escalado, rotación, transparencias, espejo, etc).Y también para poder asignar a la variable GRAPH (ya la veremos)de un proceso un texto, por ejemplo, con lo que un texto podrá asociarse a un proceso obteniendo así para ese texto todos los beneficios de la gestión de procesos (señales, colisiones, jerarquía,etc).

PARÁMETROS:
INT FUENTE : Tipo de letra
STRING TEXTO : Texto a dibujar
INT ALINEACIÓN : Tipo de alineación

VALOR DE RETORNO: INT : Identificador del nuevo gráfico

Como ejemplo de utilización de **write_in_map()**, puedes probar de ejecutar este código:

```
Import "mod_video";
Import "mod_screen";
Import "mod_text";
Import "mod_key";
Import "mod_map";

Process Main()
Private
      int idgraf;
```

```
End
Begin
    set_mode(640,480,32);
    /* Se crea un mapa nuevo con el texto "Esto es un ejemplo...1,2,3"  con el punto de
control en el centro, y se asigna como grafico de fondo*/
    idgraf = write_in_map(0,"Esto es un ejemplo...1,2,3",4);
    put_screen(0, idgraf);
    Repeat
        if(key(_a)) clear_screen();end
        if(key(_s)) put_screen(0,idgraf);end
        Frame;
    Until (key(_esc));
    unload_map(0,idgraf);
End
```

**MAP_INFO(LIBRERÍA, GRÁFICO, TIPO)**

Esta función, definida en el módulo "mod_map", permite consultar en tiempo de ejecución determinadas características de un gráfico (dimensiones, coordenadas de su centro, etc), y también de su animación si éste es un gráfico animado: GIF,MNG...(estas animaciones son automáticas, e independientes de los frames por segundo establecidos por la función **set_fps()** )

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico dentro de la librería
INT TIPO : Tipo de característica a consultar. Pueden ser:
G_WIDTH :Ancho en pixels del gráfico. Equivale a valor numérico 0
G_HEIGHT :Alto en pixels del gráfico. Equivale al valor numérico 1
G_CENTER_X :Coordenada horizontal del centro. Equivale al valor numérico 2
G_CENTER_Y :Coordenada vertical del centro. Equivale al valor numérico 3
G_DEPTH :Profundidad de color (en bits por pixel). Equivale al valor numérico 5
G_FRAMES Total de "versiones" del gráfico. Equivale al valor numérico 6
G_ANIMATION_STEP Frame actual de la animación. Equivale al valor numérico 7
G_ANIMATION_STEPS Total de frames de la animación. Equivale al valor numérico 8
G_ANIMATION_SPEED Velocidad actual de la animación. Equivale a 9
G_PITCH Diferencia en bytes, en memoria, entre 2 filas del gráfico. Equivale a 4

VALOR DEVUELTO: INT : Valor consultado

**MAP_PUT ( LIBRERÍA, GRÁFICO, GRÁFICO-ORIGEN, X, Y )**

Esta función, definida en el módulo "mod_map", dibuja un gráfico directamente sobre otro, utilizando unas coordenadas dentro del gráfico de destino, (como si éste fuese el fondo de pantalla), de manera que el primer gráfico quedará integrado parte constituyente del segundo. El punto especificado con los parámetros equivaldrá al lugar dentro del gráfico destino donde se dibuje el centro del gráfico origen. Si se especifica como librería el número 0 y como gráfico el número 0, el gráfico-origen se dibujará sobre el fondo de la pantalla.

Una limitación de esta función es que ambos gráficos deben pertenecer a la misma librería. Habitualmente los gráficos de destino suelen ser gráficos temporales creados por **new_map()** , por lo que puede usarse como gráfico de origen una copia **map_clone()** -ver más abajo- de un gráfico de cualquier librería para paliar la limitación.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico de destino dentro de la librería
INT GRÁFICO-ORIGEN : Número de gráfico a dibujar dentro de la librería
INT X : Coordenada horizontal de gráfico donde se situará el centro horizontal de gráfico-origen
INT Y : Coordenada vertical de gráfico donde se situará el centro horizontal de gráfico-origen

Un ejemplo sería éste (se necesitan dos archivos png: “a.png” -relativamente grande, ya que hará de gráfico destino- y “b.png” -relativamente pequeño, ya que se incrustará dentro de "a.png"-). Como podemos ver en el

código, se carga "a.png" y dentro de él se "incrusta" el gráfico "b.png", alterándose en el proceso la imagen referenciada por *idgraf* ( es decir *idgraf* al principio apuntaba a "a.png", peró después de **map_put()** apunta al gráfico modificado mezcla de los dos, esto es importante tenerlo en cuenta). La incrustación la hemos hecho en este caso en las coordenadas (100,100) de "a.png" (contando a partir de su esquina superior izquierda). Finalmente, se muestra el resultado (es decir, la combinación de los dos gráficos) como fondo de pantalla.

```
Import "mod_video";
Import "mod_screen";
Import "mod_key";
Import "mod_map";

Process Main()
Private
 int idgraf;
End
Begin
 set_mode(640,480,32);
 idgraf=load_png("a.png");
 map_put(0,idgraf,load_png("b.png"),100,100);
 put_screen(0,idgraf);
 repeat
    frame;
 until(key(_esc))
end
```

Otro ejemplo parecido es el siguiente, en el cual se realizan cien incrustaciones seguidas, en coordenadas aleatorias dentro de un determinado límite, y se muestra el resultado final. Este nuevo gráfico formado por las múltiples incrustaciones de "b.png" en "a.png", si nos gusta y quisiéramos, lo podríamos grabar en disco,cosa que haremos cuando sepamos cómo hacerlo.

```
Import "mod_video";
Import "mod_screen";
Import "mod_key";
Import "mod_map";
Import "mod_rand";

Process Main()
Private
 int idgraf;
 int i;
End
Begin
 set_mode(640,480,32);
 idgraf=load_png("a.png");
 for(i=0;i<100;i++)
           map_put(0,idgraf,load_png("b.png"),rand(0,200),rand(0,200));
 end
 put_screen(0,idgraf);
 repeat
    frame;
  until(key(_esc))
end
```

| **MAP_XPUT ( LIBRERÍA, GRÁFICO, GRÁFICO-ORIGEN, X, Y, ÁNGULO, TAMAÑO, FLAGS, REGION )** |
|---|
| Esta función, definida en el módulo "mod_map", dibuja un gráfico directamente sobre otro, utilizando unas coordenadas dentro del gráfico de destino, (como si éste fuese el fondo de pantalla), de manera que el primer gráfico quedará integrado parte constituyente del segundo. El punto especificado con los parámetros equivaldrá al lugar donde se dibuje el centro del gráfico a dibujar. Sin embargo, a diferencia de **map_put()** , esta función tiene la posibilidad de modificar el ángulo de orientación, el tamaño y/o el recorte de región que se quiera que tenga el gráfico-origen en el momento de su incrustación.<br><br>Una limitación de esta función es que ambos gráficos deben perteneces a la misma librería. Esto no suele ser problema porque habitualmente los gráficos de destino suelen ser gráficos temporales creados por **new_map()** , por lo |

que puede usarse como gráfico de origen una copia **map_clone()** (ya la veremos) de un gráfico de cualquier librería para paliar la limitación. También se puede utilizar **map_xputnp()** si se desean utilizar dos gráficos de diferentes librerías y obtener una funcionalidad similar.

Los flags son una suma ("+") o unión con el operador OR ("|") de alguno de los siguientes valores:

- 1: Dibuja el gráfico reflejado horizontalmente, como visto en un espejo.
- 2: Dibuja el gráfico reflejado verticalmente.
- 4: Dibuja el gráfico con transparencia del 50%.
- 128: Dibuja el gráfico sin tener en cuenta los pixels transparentes del mismo. Esta opción no es compatible con el flag 4.

Para consultar todos los valores posibles, incluyendo sus correspondencias uno a uno con determinadas constantes predefinidas de Bennu y varios ejemplos de aplicación, consultar más adelante en el siguiente capítulo el apartado correspondiente a la variable local predefinida ***FLAGS***

La región corresponde a una limitación de coordenadas (se hablará de ellas en un capítulo posterior). Hay que tener en cuenta que normalmente una región se define respecto a la pantalla, y en el caso de esta función, los mismos límites se aplican a un gráfico de destino potencialmente más pequeño. Seguramente sea preciso definir nuevas regiones para usarlas específicamente con esta función.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico de destino dentro de la librería
INT GRÁFICO-ORIGEN : Número de gráfico a dibujar dentro de la librería
INT X : Coordenada horizontal
INT Y : Coordenada vertical
INT ÁNGULO : Ángulo del gráfico a dibujar en miligrados (90000 = 90º)
INT TAMAÑO : Tamaño del gráfico a dibujar, en porcentaje (100 = tamaño original)
INT FLAGS : Flags para el dibujo
INT REGION : Región de corte (0 = ninguna)

Un ejemplo sería el siguiente (se necesitan dos archivos png: "a.png" -preferiblemente grande- y "b.png" -preferiblemente pequeño- ). En este ejemplo se "incrusta" dentro del gráfico "a.png" el gráfico "b.png", a cada frame y en distintas posiciones y con distinto ángulo, tamaño y flags. Este nuevo gráfico formado por las múltiples incrustaciones de "b.png" en "a.png", si quisiéramos, lo podríamos grabar en disco,cosa que haremos cuando sepamos cómo hacerlo:

```
Import "mod_video";
Import "mod_screen";
Import "mod_key";
Import "mod_map";
Import "mod_rand";

Process Main()
Private
 int idgraf;
 int i;
End
Begin
 set_mode(640,480,32);
 idgraf=load_png("fondo.png");
 for(i=0;i<100;i++)
 map_xput(0,idgraf,load_png("b.png"),rand(0,639),rand(0,479),rand(0,360000),rand(5,100),rand(1,2));
 end
 put_screen(0,idgraf);
 repeat
    frame;
  until(key(_esc))
end
```

**MAP_XPUTNP (LIBRERIA,GRAFICO,LIBRERIA-ORIGEN,GRAFICO-ORIGEN,X,Y,ÁNGULO, ESCALA-X, ESCALA-Y,FLAGS)**

Esta función, definida en el módulo "mod_map", dibuja un gráfico (gráfico de origen) directamente sobre otro

(gráfico de destino). Si los parámetros avanzados no son necesarios, se puede usar **map_put()** o **map_xput()** en su lugar. Si el ángulo es 0 y el tamaño es 100, la velocidad será mayor ya que el gráfico no necesita rotarse o escalarse.

Una gran diferencia práctica con **map_xput()** es que **map_xputnp()** permite (a diferencia de la primera) especificar como gráfico-origen y gráfico-destino dos gráficos pertenecientes a DIFERENTES librerías.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG que contiene el gráfico de destino
INT GRÁFICO : Número de gráfico de destino dentro de la librería
INT LIBRERÍA-ORIGEN : Número de librería FPG que contiene el gráfico de origen.
INT GRÁFICO-ORIGEN : Número de gráfico a dibujar dentro de la librería
INT X : Coord. horizontal del gráfico de destino donde se colocará el gráfico de origen
INT Y : Coord. vertical del gráfico de destino donde se colocará el gráfico de origen
INT ÁNGULO : Ángulo del gráfico a dibujar en miligrados (90000 = 90º)
INT ESCALA-X: Escala que define en el eje horizontal el tamaño del gráfico de origen
INT ESCALA-Y: Escala que define en el eje vertical el tamaño del gráfico de origen
INT TAMAÑO : Tamaño del gráfico a dibujar, en porcentaje (100 = tamaño original)
INT FLAGS : Flags para el dibujo

Un ejemplo de esta función podría ser:

```
Import "mod_video";
Import "mod_screen";
Import "mod_key";
Import "mod_map";
Import "mod_rand";

Process Main()
Private
 int destgraph;
 int origgraph;
End
Begin
 set_mode(640,480,32);
 // El gráfico de destino será un cuadrado rojo
 destgraph=new_map(400,400,32);
 map_clear(0,destgraph,rgb(255,0,0));
 //El gráfico de origen será un cuadrado azul
 origgraph=new_map(100,100,32);
 map_clear(0,origgraph,rgb(0,0,255));
 repeat
    if(key(_a))
/* Dibuja el cuadrado azul en el centro del cuadrado rojo (200,200) transparentemente (4)
en un tamaño horizontal  y vertical y ángulo aleatorios*/
       map_xputnp(0,destgraph,0,origgraph,200,200,rand(-180000,180000),rand(50,150),rand(50,150),4);
    end
// Muestra el resultado final.La línea siguiente es equivalente a put_screen(0,destgraph)
    map_put(0,0,destgraph,320,240);
    frame;
 until(key(_esc))
end
```

**PUT(LIBRERIA,GRAFICO,X,Y)**

Esta función, definida en el módulo "mod_screen", dibuja un gráfico en el fondo de pantalla, en las coordenadas especificadas como parámetro. El punto especificado equivaldrá al lugar donde se dibuje el centro del gráfico. Por lo tanto, añade un poco más de flexibilidad que **put_screen()**, la cual siempre coloca el centro del gráfico en el centro de la ventana del juego.

La misma funcionalidad se puede obtener con **map_put()** y sus dos primeros parámetros iguales a 0.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

| |
|---|
| INT GRÁFICO : Número de gráfico a dibujar<br>INT X : Coordenada horizontal<br>INT Y : Coordenada vertical |

Como ejemplo de esta función, en vez de poner un fondo “estándar”, vamos a ser más originales y vamos a poner un fondo en mosaico. Se repetirá horizontal y verticalmente una imagen (llamado en este ejemplo "icono.png"; lo importante es que ha de ser de dimensiones bastante menores que la ventana) de manera que se cubra toda la extensión de éste, obteniendo ese efecto mosaico.

```
Import "mod_video";
Import "mod_screen";
Import "mod_key";
Import "mod_map";

Process Main()
Private
            int idpng;
            int ancho;
            int alto;
            int i,j;

End
Begin
  set_mode(800,600,32);
  idpng=load_png("icono.png");
  ancho=map_info(0,idpng,g_width);
  alto=map_info(0,idpng,g_height);
/*El truco está en repetir la orden put a lo largo y ancho de la ventana,mediante dos
bucles for: el primero la irá recorriendo "columna" a "columna" y el segundo, para cada
una de esas "columnas", todas las "filas". Como las coordenadass donde la función put
pinta -o sea,el tercer y cuarto parámetro-corresponden al centro del gráfico en cuestión,
los valores de los dos bucles tendrán que valer las coordenadas de las diferentes
posiciones del centro de ese gráfico a lo largo de toda la ventana para cubrirla entera,
y sin solaparse. Por eso,el primer valor del primer bucle es "ancho/2", porque la
coordenada horizontal del centro del primer gráfico a dibujar (el de la esquina superior
izquierda) és esa, y por eso el primer valor del segundo bucle es "alto/2", por la misma
razón.Para buscar la coordenada del siguiente centro es fácil: si se busca el siguiente
centro horizontal, es sumar el ancho del gráfico a la posición del centro actual, y
similarmente si buscamos el siguiente centro vertical: sumando la posición del alto del
gráfico a la posición del centro actual. Ésos son los valores del paso de los dos bucles.
Y el significado del la condición de salida de los fors se basa en detectar cuándo ya no
hace falta seguir pintando gráficos tanto horizontalmente como verticalmente pose se ha
llegado al final de la ventana. La condición viene de que si la coordenada del siguiente
centro es mayor que el extremo de la ventana (más un último ancho/alto para forzar a que
en los límites de la ventana se pinten los gráficos aunque estén recortados) ya no hace
falta seguir pintando por allí.Este sistema de hacer mosaicos en un tema posterior, en el
último capítulo, se tratará más en profundidad, ya que es bastante importante para el uso
de los llamados "tiles"*/
for(i=ancho/2;i<800+ancho;i=i+ancho)
          for(j=alto/2;j<600+alto;j=j+alto)
                    put(0,idpng,i,j);
/*Ya sabes que si aquí pusiéramos una línea frame, veríamos la construcción del mosaico
paso a paso, "en tiempo real".*/
          end
end
repeat
          frame;
until(key(_esc))
end
```

A continuación muestro un ejemplo de la función **map_clear()** en realidad, pero no se puso cuando se explicó ésta porque se hace uso también de la función recién conocida **put()**. Este código simplemente dibuja un cuadrado verde que cambia progresivamente de nivel de transparencia de mínimo a máximo y viceversa alternativamente.

```
import "mod_map";
```

```
import "mod_key";
import "mod_video";
import "mod_screen";

Process Main()
Private
    int a=0;
    int da=10;
    int idgraf;
Begin
    set_mode(320,240,32);
    idgraf = new_map(100,100,32);
    Repeat
        a = a + da;
        if(a<0)
            da = -da;
            a = 0;
        elseif(a>255)
            da = -da;
            a = 255;
        end
        map_clear(0,idgraf,rgba(0,255,0,a));
        clear_screen();
        put(0,idgraf,160,120);
        frame;
    Until(key(_esc))
End
```

¿Y por qué se hace uso de la función **put()** cuando se podría haber utilizado perfectamente **put_screen()** ? Pues porque parece ser, según las pruebas realizadas, que **put_screen()** no tiene en cuenta el factor alfa y siempre se ve el cuadrado verde opaco, por lo que hay que recurrir a **put()**, que sí lo tiene en cuenta.

Por otro lado, fijarse que siempre, antes de un **put()**, se realiza un **clear_screen()**. Este detalle es importante para que podamos ver que el cuadro desaparece y vuelve a aparecer. Si no lo pusiéramos, el fondo de pantalla se iría pintando sobre el anterior, con lo que cuando tuviéramos el cuadrado verde pintado por primera vez, aunque los siguientes fondos fueran cuadrados más transparentes, al permanecer pintados los fondos anteriores, siempre veríamos sus restos: es decir, el cuadradado verde siempre ahí.

Una idea que se podría ocurrir donde se utilizaría la función **put()** sería la realización de un movimiento de scroll automático en el gráfico del fondo de la pantalla, modificando convenientemente dentro de algún bucle los dos últimos parámetros de **put()**. No obstante, existe una manera mucho más óptima, versátil y potente para realizar scrolls, con la función **start_scroll()** y familia, que ya estudiaremos más adelante.

**XPUT(LIBRERIA, GRAFICO,X,Y,ANGULO,TAMAÑO,FLAGS,REGION):**

Esta función, definida en el módulo "mod_screen", dibuja un gráfico en el fondo de pantalla, en las coordenadas especificadas como parámetro, y aplicando todas las posibilidades de dibujo de un proceso (ángulo de rotación,transparencias, escalado...). El punto especificado equivaldrá al lugar donde se dibuje el centro del gráfico, y a partir de cual se rotará o cambiará de tamaño.

La misma funcionalidad se puede obtener con **map_xput()** y sus dos primeros parámetros iguales a 0

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico a dibujar
INT X : Coordenada horizontal
INT Y : Coordenada vertical
INT ÁNGULO : Ángulo en miligrados (90000 = 90º)
INT TAMAÑO : Tamaño en porcentaje (100 = tamaño original)
INT FLAGS : Flags para el dibujo
INT REGION : Región de recorte (0 = ninguna)

Un ejemplo trivial para ver su utilidad (necesitarás un gráfico llamado “icono.png”):

```
Import "mod_video";
Import "mod_key";
Import "mod_map";
Import "mod_screen";

Process Main()
Private
           int idpng;
End
Begin
  set_mode(800,600,32);
  idpng=load_png("icono.png");
  xput(0,idpng,120,240,0,50,1,0);
  xput(0,idpng,520,240,90000,200,4,0);
  //Como se pinta en la región 1, y ésta no existe, este gráfico no se ve.
  xput(0,idpng,60,400,180000,100,0,1);
  loop
         if(key(_esc)) break; end
         frame;
  end
end
```

| **MAP_CLONE ( LIBRERÍA, GRÁFICO )** |
|---|
| Esta función, definida en el módulo "mod_map", crea una copia en memoria de un gráfico cualquiera, un "clon". El nuevo gráfico se crea como perteneciente a la librería 0, y Bennu asigna un número nuevo disponible para el mismo. El valor de retorno de esta función corresponde al número del nuevo gráfico, y debe recogerse para poder utilizarse en cualquier operación gráfica.<br><br>Si lo que se desea es crear un gráfico nuevo dentro de una librería FPG distinta de la del sistema, entonces es preferible usar la función **fpg_add()**<br><br>PARÁMETROS:<br><br>INT LIBRERÍA : Número de librería FPG<br>INT GRÁFICO : Número de gráfico dentro de la librería |

Un ejemplo:

```
Import "mod_video";
Import "mod_key";
Import "mod_map";
Import "mod_text";

Process Main()
Private
  int orimap,clonemap,clonemap2;
End
Begin
  set_mode(320,240,32);
  write(0,80,50,4,"Original ");
  orimap=new_map(32,20,32);
  map_clear(0,orimap,rgb(127,127,127));
  map_put(0,0,orimap,80,100);
  write(0,240,50,4,"Clon 1");
  clonemap=map_clone(0,orimap);
  map_put(0,0,clonemap,240,100);
  write(0,400,50,4,"Clon 2");
  clonemap2=map_clone(0,clonemap);
  map_put(0,0,clonemap2,400,100);
  repeat
    frame;
  until(key(_esc))
end
```

Una utilidad práctica de esta función es partir de un gráfico inicial, el cual se clona, y modificar el gráfico clonado como se desee (por ejemplo con las funciones de la familia *map_put*...), de tal forma que obtengamos un nuevo gráfico, levemente diferente al original en tiempo real sin tener la necesidad de dibujarlo explícitamente y guardarlo en un FPG como un elemento más,por ejemplo. El siguiente código muestra esto:

```
Import "mod_video";
Import "mod_key";
Import "mod_map";
Import "mod_text";

Process Main()
Private
  int grancuad,pequecuad,cloncuad,cloncuad2;
End
Begin
  set_mode(320,240,32);
  //Genero el cuadrado grande amarillo (pero no lo muestro todavía)
  grancuad=new_map(50,50,32);
  map_clear(0,grancuad,rgb(255,255,0));
  //Genero el cuadrado pequeño rojo (pero no lo muestro todavía)
  pequecuad=new_map(10,10,32);
  map_clear(0,pequecuad,rgb(255,0,0));
  //Dibujo en pantalla el cuadrado amarillo original
  map_put(0,0,grancuad,80,100);

  //Genero una copia en memoria del cuadrado amarillo...
  cloncuad=map_clone(0,grancuad);
  //...y sobre él pinto el cuadrado rojo...
  map_put(0,cloncuad,pequecuad,20,20);
  //...y muestro el resultado en pantalla
  map_put(0,0,cloncuad,180,100);

  //Repito el proceso a partir del primer clon, para crear un segundo
  cloncuad2=map_clone(0,cloncuad);
  map_put(0,cloncuad2,pequecuad,40,40);
  map_put(0,0,cloncuad2,280,100);

  repeat
    frame;
  until(key(_esc))
end
```

Esta misma idea es la que se suele implementar con procesos (en el capítulo siguiente aprenderemos qué son y cómo funcionan), de manera que podamos tener elementos de nuestro juego (personajes, escenarios, ítems...) cuyos gráficos se hayan generado completamente en memoria, a partir de elementos dispares incrustados. Un ejemplo sería por ejemplo tener un gráfico de nuestro protagonista y un gráfico de una mancha de sangre, e ir modificando en nuestro juego el gráfico de nuestro protagonista añadiéndole más heridas sobre él a medida que fuera perdiendo más y más vida. Incluso podríamos tener código centralizado que se dedicara de forma exclusiva e independiente a realizar este tipo de modificaciones de gráficos "al aire", utilizando rutinas propias (hechas por nosotros) especializadas en esta generación de imágenes para nuestro juego.

Finalmente, como ejercicio, intenta entender lo que ocurre cuando ejecutamos el siguiente código (necesitarás una imagen llamada “dibujo.png”):

```
Import "mod_video";
Import "mod_key";
Import "mod_map";
Import "mod_screen";

Process Main()
Private
  int graforig, grafdest, grafintermedio;
End
Begin
set_mode(640,480,32);
```

```
graforig=load_png("dibujo.png");
put_screen(0,graforig);
/*Todo lo que viene a continuación hasta el loop es un código que transforma graforig en
grafdest, utilizando grafintermedio como paso intermedio. Para ello clonamos
grafintermedio a partir de graforig, creamos grafdest con el mismo tamaño que graforig
pero sin contenido e incrustamos grafintermedio (aumentando su tamaño el doble del
original) en grafdest. Finalmente, descargamos grafintermedio porque ya no lo
necesitamosmás. Se podría haber encapsulado en un "function", pero esto no lo hemos visto
todavía */
grafintermedio=map_clone(0,graforig);
grafdest=new_map(map_info(0,graforig,g_width),map_info(0,graforig,g_height),32);
map_xput(0,grafdest,grafintermedio,0,0,0,200,0);
unload_map(0,grafintermedio);
repeat
          if(key(_o)) put_screen(0,graforig); end
          if(key(_d)) put_screen(0,grafdest);; end
          frame;
until (key(_esc))
end
```

**MAP_BLOCK_COPY ( LIBRERÍA, GRÁFICO, X-DESTINO, Y-DESTINO, GRÁFICO-ORIGEN, X-ORIGEN, Y-ORIGEN, ANCHO, ALTO, FLAGS )**

Esta función, definida en el módulo "mod_map", copia una sección de un gráfico delimitada por unas coordenadas de origen (para la esquina superior izquierda de la zona), un ancho y un alto, a un gráfico de destino, posiblemente en otras coordenadas.

Gracias al valor de flags indicado es posible determinar si los píxeles transparentes en el gráfico de origen se ignoran (flags 0) o se copian también (flags 128).

La única limitación es que ambos gráficos deben formar parte de la misma librería. Si se desea realizar la operación con gráficos en librerías diferentes, la alternativa es crear una copia de uno de los dos gráficos mediante **fpg_add()** o **map_clone()**

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico de destino dentro de la librería
INT X-DESTINO : Coordenada horizontal de destino
INT Y-DESTINO : Coordenada vertical de destino
INT GRÁFICO-ORIGEN : Número de gráfico de origen dentro de la librería
INT X-ORIGEN : Coordenada horizontal dentro del gráfico de origen
INT Y-ORIGEN : Coordenada vertical dentro del gráfico de origen
INT ANCHO : Ancho en píxeles del bloque a copiar
INT ALTO : Alto en píxeles del bloque a copiar
INT FLAGS : Valor de flags

Un ejemplo autoexplicativo:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_screen";

process main()
private
  int graf1,graf2,grafdest;
  //Ancho de la imagen sobre la cual se copiará -parte de- otra imagen diferente ("fuente")
  int anchoDestino=50;
  //Alto de la imagen sobre la cual se copiará -parte de- otra imagen diferente ("fuente")
  int altoDestino=50;
  int anchoFuente=25; //Ancho de la imagen que será copiada sobre otra imagen ("destino")
  int altoFuente=25; //Alto de la imagen que será copiada sobre otra imagen ("destino")
end
begin
```

```
  set_mode(640,480,32);
  set_fps(20,1);
  //Creo y visualizo imagen "destino": cuadrado de color rojo de 50x50
  graf1 = new_map(anchoDestino,altoDestino,32); map_clear(0,graf1,rgb(255,0,0));
  put_screen(0,graf1);
  /*Creo (sin visualizar)la imagen "fuente" que se copiará -al menos, parte- sobre
  "destino": cuadrado de color amarillo de 25x25 */
  graf2= new_map(anchoFuente,altoFuente,32);map_clear(0,graf2,rgb(255,255,0));
  while (!key(_a)) frame;end //Hasta que no pulse la tecla Space, no continúo

 /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida con
 la coordenada (0,0) de "destino",y copio todo el gráfico de "fuente", sin ningún flag*/
  grafdest=map_clone(0,graf1);
  map_block_copy(0,grafdest,0,0,graf2,0,0,anchoFuente,altoFuente,0);
  put_screen(0,grafdest);
  while (!key(_s)) frame;end

 /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida con
 la coordenada (anchoDestino/2,altoDestino/2) de "destino",y copio todo el gráfico de
 "fuente", sin ningún flag*/
  grafdest=map_clone(0,graf1);
  map_block_copy(0,grafdest,anchoDestino/2,altoDestino/2,graf2,0,0,anchoFuente,altoFuente,0);
  put_screen(0,grafdest);
  while (!key(_d)) frame;end

  /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida
  con la coordenada (anchoDestino/2,0) de "destino",pero copio sólamente una parte del
  gráfico de "fuente" (en concreto, un cuarto de la imagen original ya que el bloque está
  definido por (anchoFuente/2,altoFuente/2)), sin ningún flag*/
  grafdest=map_clone(0,graf1);
  map_block_copy(0,grafdest,anchoDestino/2,0,graf2,0,0,anchoFuente/2,altoFuente/2,0);
  put_screen(0,grafdest);
  while (!key(_f)) frame;end

  /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida
  con la coordenada (0,altoDestino/2) de "destino",pero copio sólamente una parte del
  gráfico de "fuente" (en concreto, un noveno de la imagen original ya que el bloque está
  definido por (anchoFuente/3,altoFuente/3)), con un flag de transparencia*/
  grafdest=map_clone(0,graf1);
  map_block_copy(0,grafdest,0,altoDestino/2,graf2,0,0,anchoFuente/3,altoFuente/3,4);
  put_screen(0,grafdest);

  loop if(key(_esc)) break;end frame; end
end
```

A continuación pongo otro código de ejemplo similar al anterior, pero en donde las copias de los bloques de la imagen “fuente” se realizan de forma aleatoria, tanto en posición como en tamaño, consiguiendo un efecto curioso, al menos. Para poderlo ejecutar se necesitarán dos imágenes PNG de nombre “a.png” y “b.png” (preferiblemente grandes)

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_screen";
Import "mod_rand";

process main()
Private
   int x_destino, y_destino;
   int x_origen, y_origen;
   int ancho, alto;
   int png1, png2;
end
Begin
    set_mode(640,480,32);
    png1=load_png("sol.png");
    png2=load_png("fondo.png");
Loop
/*Definimos el punto de la segunda imagen a partir del cual,estableciendo su alto y ancho
```

```
también, crearemos el bloque que posteriormente copiaremos sobre la primera imagen.
Evidentemente, la  posición inicial ha de pertenecer a la segunda imagen,por lo que su
valor ha de estar dentro del tamaño de ésta (en nuestro caso, hemos elegido una imagen de
100x100). */
   x_origen=rand(0,100);
   y_origen=rand(0,100);
   ancho=rand(1, 64);
   alto=rand(1, 64);
/*Definimos la posición dentro de la primera imagen a partir de la cual se colocará el
bloque establecido en las líneas anteriores.Evidentemente, esta posición ha de pertenecer
a la primera imagen, por lo que su valor ha de estar dentro del tamaño de ésta (en
nuestro caso, hemos elegido una imagen de 640x480)*/
   x_destino=rand(0, 640);
   y_destino=rand(0, 480);
/*Ésta es la orden que realiza todo el proceso explicado hasta ahora.De hecho, lo único
que habíamos hecho hasta aquí era declarar variables,que son las que se utilizan en este
comando*/
   map_block_copy(0,png1, x_destino, y_destino, png2, x_origen, y_origen, ancho, alto,0);
//Visualizamos el resultado
   put_screen(0, png1);
  Frame;
  If(key(_esc)) Break; End
End
End
```

Con las funciones vistas hasta ahora se pueden lograr efectos visuales muy interesantes, como por ejemplo realizar ondulaciones de una imagen, o bien transiciones de diferentes tipos entre imágenes. Como los ejemplos que muestran estos efectos son de un nivel un poco más elevado del que tenemos a estas alturas de manual, se han relegado al último capítulo, donde aparece un apartado con códigos que implementan ondulaciones y otro con códigos que implementan transiciones. Remito al lector con conocimientos avanzados de Bennu a consultar estos códigos y poderlos implementar en sus propios proyectos.

Otras funciones interesantes son:

| **FPG_EXISTS( LIBRERÍA)** |
|---|
| Esta función, definida en el módulo "mod_map", devuelve 1 si una librería FPG existe con el código indicado. Puede haber hasta 1000 librerías FPG en el programa, con códigos de 0 a 999. Es posible usar esta función dentro de un bucle para enumerar todos los gráficos que existan en memoria, con la ayuda de **map_exists()**.<br><br>PARÁMETROS: INT LIBRERÍA : Número de librería FPG |

| **MAP_EXISTS(LIBRERÍA, GRÁFICO)** |
|---|
| Esta función, definida en el módulo "mod_map", devuelve 1 si un gráfico dado existe dentro de la librería FPG indicada. Dado que sólo puede haber 1000 gráficos en una librería, con códigos de 0 a 999, es posible usar esta función dentro de un bucle para enumerar todos los gráficos que existan dentro de una librería FPG.<br><br>PARÁMETROS:<br><br>INT LIBRERÍA : Número de librería FPG<br>INT GRÁFICO : Número de gráfico dentro de la librería |

| **GET_SCREEN()** |
|---|
| Esta función, definida en el módulo "mod_screen", crea un nuevo gráfico con el estado de pantalla actual. Es decir, realiza una captura de pantalla. Más concretamente, crea un nuevo gráfico con el tamaño y número de bits por pixel equivalentes a la pantalla actual (como si fuera **new_map()**) y seguidamente dibuja sobre él el fondo de pantalla y todos los procesos, textos y objetos, en el mismo orden que existan. El resultado es que el nuevo gráfico contendrá una representación fiel de la última pantalla que fue mostrada al usuario.<br><br>La imagen de la captura queda almacenada en memoria, y lo que devuelve **get_screen()** es su código identificador |

(igual que como lo hace **new_map()**, por ejemplo) el cual es creado en ese momento con un valor aleatorio mayor de 999. Usando dicho código identificador, podremos usar esa imagen para cualquier cosa: asignarla como imagen de un proceso, guardarla en disco, etc.

VALOR DE RETORNO: INT : Código del nuevo gráfico

El nuevo gráfico obtenido por **get_screen()** utiliza el estado actual de todos los procesos y objetos, lo cual significa que cualquier proceso que se ejecute antes que el proceso que haga **get_screen()** puede mostrar un aspecto ligeramente diferente al de la pantalla dibujada el frame anterior. Esto puede resultar útil, ya que por ejemplo es posible obtener una copia de la pantalla sin el cursor del ratón simplemente eliminando su gráfico antes de llamar a **get_screen()** y volviéndolo a restaurar inmediatamente después (para que se vea normalmente en el próximo frame, sin apreciarse ningún "parpadeo").

Si se piensa realizar varias capturas de pantalla, se ha de tener en cuenta el descargar de la memoria la captura (*unload_map(0,captura)*) ya que cada **get_screen()** no sobreescribe el anterior sino que lo guarda en otro sitio de la memoria con otro código aleatorio, por lo que si se realizan muchas capturas sin descargar, el programa rápidamente empezará a ir cada vez más lento ya que se estará consumiendo los recursos -la memoria- del ordenador de forma incontrolada.

Para grabar las capturas en el disco duro, no funciona hacer un **save_fpg()** (función que veremos posteriormente) del fpg 0 (la librería del sistema), ya que los graficos de indice mayor de 999 no se graban: se tendría que cambiar el código del gráfico de la captura por otro que fuera menor de 999 y no estuviera ocupado, con lo que el trabajo del programador aumenta. Donde no hay problema es utilizando la función **save_png()**.

**SAVE_PNG(LIBRERIA,GRAFICO,"FICHERO")**

Esta función, definida en el módulo "mod_map", guarda en disco una imagen en formato PNG, a partir de otra imagen cargada previamente en memoria .

PARAMETROS:

INT LIBRERIA: Código de la librería FPG a partir de la cual se obtiene el gráfico origen – o bien 0 si se trabaja con la librería del sistema-.
INT GRAFICO : Número del gráfico dentro de ese FPG, o bien el código obtenido mediante **new_map()**, **load_png()** o similares.
STRING "fichero" : Ruta completa del gráfico PNG a guardar.

En combinación con otros comandos, como ***map_xput()*** o ***map_xputnp(), save_png()*** nos ayudará a guardar en disco imágenes generadas y manipuladas en nuestros programas que pueden mostrar efectos visuales interesantes. He aquí, por ejemplo, un código artístico. Éste guarda (después de haber cargado un gráfico desde un FPG llamado "graficos.fpg" y haberlo puesto como fondo) en un fichero temporal en memoria una captura del fondo hecha a cada frame con **save_png()** ,y con **map_xput()**, pega este PNG creado encima de la imagen de fondo, girado 0.1 grados sobre su centro y un 1% más pequeño. Se consigue un efecto bonito.

```
Import "mod_video";
Import "mod_map";
Import "mod_screen";
Import "mod_key";

process main()
private
          int fichero1;
          int a=0;
end
begin
          fichero1 = load_fpg("graficos.fpg");
          set_mode(320,240,32);
          set_fps(150,0);
          put_screen(fichero1,1);
          repeat
                    //Se guarda en disco el gráfico actual visible en el fondo
```

```
                    save_png(0,0,"tp_save_png.png");
                    a=a+100;
    /*Se obtiene un "clon" del PNG actualmente grabado, se reescala un -1% y se gira una
    cantidad "a" y seguidamente se vuelve a incrustar en el fondo*/
                    map_xput(0,0,load_png("tp_save_png.png"),160,120,a,99,7);
                    frame;
          until (key(_esc))
end
```

El fichero PNG creado no se borra cuando se cierra el programa. Aviso: no dejes funcionando mucho rato el ejemplo que ralentiza la máquina.

Un ejemplo de **get_screen()** combinado con **save_png()** sería el siguiente código, donde grabamos en un fichero la captura de pantalla realizada:

```
Import "mod_video";
Import "mod_map";
Import "mod_screen";
Import "mod_key";

process main()
private
          int captura;
          int fondo;
end
begin
          set_mode(320,240,32);
          fondo=load_png("a.png");
           put_screen(0,fondo);
          repeat
          if(key(_c))
                    captura=get_screen();
                    save_png(0,captura,"captura.png");
          end
          frame;
          until (key(_esc))
end
```

Incluso podríamos utilizar funciones como **map_xput()** o **map_xputnp()** para grabar capturas de pantallas redimensionadas, movidas de orientación, semitransparentes,etc. Por ejemplo, podríamos modificar el código anterior para obtener una fichero que guarde una captura de la mitad de tamaño que la captura original.

```
Import "mod_video";
Import "mod_map";
Import "mod_screen";
Import "mod_key";

process main()
private
           int original;
          int captura;
           int nuevo;
end
begin
          set_mode(320,240,32);
           original=load_png("fondo.png");
           put_screen(0,original);
          repeat
          if(key(_c))
                          //Se obtiene la captura de pantalla
                          captura=get_screen();
       /*Se genera una nueva imagen vacía de la mitad de tamaño que la pantalla original,
       donde se alojará un "clon" (reescalado a la mitad de la captura)*/
                          nuevo=new_map(160,120,32);
      /*Se incrusta (centrada en el punto (80,60)) la captura en la imagen vacía creada
      en la línea anterior, reescalada a la mitad para que quepa*/
                          map_xput(0,nuevo,captura,80,60,0,50,0);
```

```
                            //Y se graba esta nueva imagen en un fichero
                             save_png(0,nuevo,"nuevo.png");
            end
            frame;
            until (key(_esc))
end
```

Otra función más:

| GRAPHIC_SET(LIBRERÍA, GRÁFICO, TIPO, VALOR) |
|---|
| Esta función, definida en el módulo "mod_map", permite cambiar en tiempo de ejecución determinadas características de un gráfico, básicamente las coordenadas de su centro, y características interesantes si éste es un gráfico animado (GIF,MNG...). En éstos, puede usarse para escoger un frame concreto en los gráficos animados, o cambiar la velocidad de su animación, en milisegundos. Por ejemplo, una velocidad de 0 detiene la animación del gráfico.Las animaciones de los gráficos son automáticas, e independientes de los frames por segundo establecidos por la función **set_fps()**<br><br>PARÁMETROS:<br><br>INT LIBRERÍA : Número de librería FPG<br>INT GRÁFICO : Número de gráfico dentro de la librería<br>INT TIPO : Tipo de característica a cambiar. Pueden ser:<br>G_CENTER_X :Coordenada horizontal del centro<br>G_CENTER_Y Coordenada vertical del centro<br>G_ANIMATION_STEP Frame actual de la animación<br>G_ANIMATION_SPEED Velocidad actual de la animación<br>INT VALOR : Nuevo valor a asignar a la característica |

## *Dibujar primitivas gráficas

Por primitiva gráfica se entiende una figura simple de dibujar, como una línea, un círculo o un rectángulo. Estas formas sencillas se llaman primitivas porque a partir de ellas se pueden crear la mayoría de dibujos posibles: cualquier gráfico de hecho no es más que un conjunto más o menos complejo de círculos (y elipses) y polígonos (rectángulos, triángulos...)

Para dibujar primitivas gráficas, Bennu dispone de unas cuantas funciones específicas, todas ellas definidas en el módulo "mod_draw". Está claro que podríamos utilizar como hasta ahora cualquier editor de imágenes para dibujar los círculos y rectángulos y luego utilizar el FPG correspondiente, pero en el caso de estas figuras tan sencillas de dibujar, Bennu ofrece la posibilidad de crearlas mediante código, mejorando así la velocidad de ejecución del programa, y también la comodidad del programador, que no tiene que recurrir a dibujos externos creados previamente.

La primera función que tendremos que escribir siempre cuando queramos pintar primitivas gráficas, sea cual sea su tipo, es **drawing_map()**

| DRAWING_MAP(LIBRERÍA, GRÁFICO) |
|---|
| Esta función escoge el gráfico destino donde se pintarán las primitivas gráficas (**draw_rect()**, **draw_line()**, **draw_circle()**, etc) que vayamos a dibujar, ya que en Bennu cualquier primitiva gráfica siempre se ha de pintar sobre un gráfico determinado, definido mediante esta función. Es decir, no es válido llamar a ninguna de estas funciones si no se escoge antes un fondo como destino, pero basta con que sea simplemente un gráfico creado inmediatamente antes con **new_map()**, **write_in_map()** o similar.<br><br>El gráfico (0, 0) representa el fondo de pantalla.<br><br>PARÁMETROS: |

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico a dibujar

**DRAWING_COLOR( COLOR)**

Esta función permite escoger el color -y su nivel de transparencia, en el caso de 32 bits- con el que se dibujarán las primitivas gráficas (**draw_rect()**, **draw_box(), draw_line()**, **draw_circle(), draw_fcircle(),** etc) hasta nuevo cambio..

Como siempre, en el caso de modos gráficos de 256 colores (8 bits), el valor de color debe ser un número de 0 a 255 y en los otros modos es recomendable usar la función **rgba()** ó **rgb()** para obtener la codificación de un color concreto, o bien usar un color obtenido por una función como **map_get_pixel()**.

PARÁMETROS: INT COLOR : Número de color

**DRAWING_ALPHA(ALPHA)**

Esta función permite escoger el valor de transparencia a utilizar con todas las primitivas gráficas (es decir, las funciones que empiezan por "draw"), hasta nuevo cambio. Un valor de 255 indica opacidad completa (sin transparencia), mientras con un valor de 0 la primitiva gráfica sería invisible. Es válido usar cualquier valor intermedio (por ejemplo, 128 para especificar una transparencia del 50%). Por defecto el valor de transparencia es de 255 y las primitivas se dibujan sin transparencia.

El efecto de esta función se suma, en gráficos de 32 bits, a la propia transparencia intrínseca que pudiera tener dicho gráfico. Es decir, **drawing_alpha()** aporta un grado de transparencia general "de serie" para todas las primitivas gráficas; si luego una primitiva gráfica en concreto utiliza un color (semi)transparente para pintarse, en esa primitiva concreta los efectos de las dos transparencias (la general de **drawing_alpha()** y la propia dada por su color) se acumularán.

PARÁMETROS: INT ALPHA : Valor de opacidad, entre 0 y 255

**DRAW_RECT(X1,Y1,X2,Y2)**

Un rectángulo consta de cuatro líneas (dos verticales y dos horizontales) y no está relleno. Esta función dibuja un rectángulo entre dos puntos sobre el gráfico escogido por la última llamada a **drawing_map()**, empleando el color y transparencia elegidos con la función **drawing_color()**. Además, se utiliza el último valor especificado con **drawing_alpha()** como nivel de transparencia predeterminado para el dibujo.

El primer punto representa el vértice superior izquierdo de la figura, y el segundo punto el vértice inferior derecho.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el rectángulo (el escogido por la última llamada a **drawing_map()**). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X1 : Coordenada horizontal del primer punto
INT Y1 : Coordenada vertical del primer punto
INT X2 : Coordenada horizontal del segundo punto
INT Y2 : Coordenada vertical del segundo punto

Vamos a escribir un pequeño ejemplo que nos pinte en pantalla varios rectángulos de dimensiones,colores y transparencias diferentes:

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_rand";
Import "mod_map";
```

```
Process Main()
begin
  set_mode(640,480,32);
  drawing_map(0,0);
  repeat
      drawing_color(rgba(255,0,0,rand(0,255)));
      draw_rect(rand(0,639),rand(0,479),rand(0,639),rand(0,479));
      frame;
  until(key(_esc))
end
```

En este ejemplo lo que hemos hecho ha sido pintar rectángulos en el gráfico de fondo de pantalla –y como no hay ninguno definido, en realidad se pintan sobre el fondo negro-.

Pero con **drawing_map()** hemos dicho que se puede definir un gráfico cualquiera sobre el cual, y sólo sobre éste, se pintarán las primitivas. Para verlo, ejecuta el siguiente ejemplo, donde se ha definido un gráfico sobre el cual se pintarán unas primitivas (y que por tanto, no podrán pintarse fuera de él), de manera que éstas no se verán por todo el fondo de la pantalla sino que aparecerán visibles solamente los trozos que se pinten sobre ese gráfico determinado. Para comparar y ver la diferencia, en el mismo ejemplo también se pintan primitivas en toda la pantalla

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
Import "mod_rand";
Import "mod_screen";

Process Main()
private
  int map;
end
begin
  set_mode(640,480,32);
  set_fps(1,1); //Cada segundo se produce un frame
/*Genero el gráfico donde se van a pintar las primitivas rojas. Fuera de él éstas no se
pintarán,pero sí las azules. Este gráfico es un cuadrado de 200x200 de color blanco*/
  map=new_map(200,200,32);
  map_clear(0,map,rgb(255,255,255));
  repeat
/*Pinto una nueva primitiva (roja) dentro del cuadrado blanco. La línea clave: estamos
diciendo que las primitivas que se generen a partir de ahora (hasta nuevo aviso)sólo se
pintarán sobre el gráfico especificado por "map"*/
            drawing_map(0,map);
            drawing_color(rgba(255,0,0,255));
            draw_rect(rand(0,639),rand(0,419),rand(0,639),rand(0,419));
            //Visualizo el gráfico, y lo doy una posición dentro de la pantalla
            put(0,map,320,240);
          frame;
/*Ahora especifico que las primitivas que se generen a partir de ahora (hasta nuevo
aviso) se pintarán en cualquier punto del fondo de la pantalla (gráfico 0,0). Eso es lo
que hacen las primitivas azules*/
            drawing_map(0,0);
            drawing_color(rgba(0,0,255,255));
            draw_rect(rand(0,639),rand(0,419),rand(0,639),rand(0,419));
            frame;
  until(key(_esc))
end
```

Creo que es fácil de entender. Primero creamos un gráfico cuadrado blanco en memoria, que será donde pintaremos rectángulos rojos. Los rectángulos rojos sólo serán pintados sobre ese gráfico, y cualquier línea se sobrepase el gráfico no se verá porque estará fuera de él. Esto se hace especificando en **drawing_map()** que queremos que el gráfico "lienzo" de las primitivas sea el gráfico "map", que es el cuadrado blanco que hemos generado. Pintamos el rectángulo y finalmente, muy importante, mostramos por pantalla ese gráfico con el rectángulo rojo acabado de pintar encima, en forma de imagen de fondo. Si no escribimo **put()** o similar, todas las modificaciones que hagamos sobre el gráfico se habrán hecho en memoria y no veremos nada en la pantalla: hay que (re)pintar el gráfico cada vez que se produzca algún cambio. Finalmente, para comparar el hecho de establecer un mapa concreto como destino de las

primitivas o no, en el siguiente fotograma de dibuja una primitiva azul, pero esta vez estableciendo como gráfico de destino el fondo de la pantalla, es decir, el gráfico (0,0). Se puede ver entonces que las primitivas azules no están limitadas al cuadrado blanco sino que se dibujan por toda la pantalla. Fíjate, no obstante, que cada vez que se vuelve a pintar el cuadrado blanco con **put()** la parte de primitiva azul que lo atravesaba en el frame anterior ahora queda tapado por el nuevo repintado.

A parte de rectángulos, también podemos dibujar más cosas:

**DRAW_CIRCLE(X,Y,RADIO)**

Dibuja un círculo (sin relleno) alrededor de un punto, sobre el gráfico escogido por la última llamada a **drawing_map()**, empleando el color y transparencia elegido en la última llamada a **drawing_color()** -y/ó **drawing_alpha()**-

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el círculo (el escogido por la última llamada a **drawing_map()**). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X : Coordenada horizontal del centro
INT Y : Coordenada vertical del centro
INT RADIO : Radio en pixels

**DRAW_LINE(X1,Y1,X2,Y2)**

Dibuja una línea entre dos puntos en el gráfico escogido por la última llamada a **drawing_map()**, empleando el color y transparencia elegido en la última llamada a **drawing_color()** -y/ó **drawing_alpha()**-

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará la línea (el escogido por la última llamada a **drawing_map()**). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X1 : Coordenada horizontal del primer punto
INT Y1 : Coordenada vertical del primer punto
INT X2 : Coordenada horizontal del segundo punto
INT Y2 : Coordenada vertical del segundo punto

**DRAWING_STIPPLE(VALOR)**

Esta función permite escoger el aspecto con el que se dibujan las líneas dibujadas posteriormente con **draw_line()** o **draw_rect()**

El parámetro es un número entero de 32 bits, en el cual cada bit a 1 representa un pixel activo en cada segmento de 32 pixels de la línea. Por defecto, el valor de stipple es (en notación hexadecimal) 0FFFFFFFh -que equivale en decimal a 4294967295- ya que todos los 32 bits están a 1 , por lo que las líneas tienen la apariencia de una línea sólida. Algunos valores útiles para este parámetro son 055555555h -que equivale a 1431655765 en decimal- para dibujar una línea de puntos o 0F0F0F0Fh -que equivale a 252645135 en decimal- para dibujar una línea de rayas.

PARÁMETROS: INT VALOR : Nuevo valor de "stipple"

NOTA: Por cierto, si quieres saber más sobre lo que significa “notación hexadecimal”, consulta el siguiente artículo: http://es.wikipedia.org/wiki/Sistema_hexadecimal

**DRAW_CURVE(X1,Y1,X2,Y2,X3,Y3,X4,Y4,NIVEL)**

Dibuja una curva Bézier entre dos puntos (puntos 1 y 4), usando otros dos puntos (puntos 2 y 3) como puntos de

control de la curva. El nivel de calidad es un valor entero entre 1 y 16. A mayor nivel de calidad, más precisa será la curva, pero más costoso será dibujarla. Se recomienda un nivel de calidad entre 6 y 9.

Usa el color y transparencia elegido con **drawing_color()**, en el gráfico escogido por la última llamada a **drawing_map()**. Además, se puede utilizar el último valor especificado con **drawing_alpha()** como nivel de transparencia predeterminado para el dibujo.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará la curva (el escogido por la última llamada a **drawing_map()**). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X1 : Coordenada horizontal del primer punto
INT Y1 : Coordenada vertical del primer punto
INT X2 : Coordenada horizontal del segundo punto
INT Y2 : Coordenada vertical del segundo punto
INT X3 : Coordenada horizontal del tercer punto
INT Y3 : Coordenada vertical del tercer punto
INT X4 : Coordenada horizontal del cuarto punto
INT Y4 : Coordenada vertical del cuarto punto
INT NIVEL : Nivel de calidad de la curva

**DRAW_BOX(X1,Y1,X2,Y2)**

Dibuja un rectángulo con relleno de color entre dos puntos sobre el gráfico escogido por la última llamada a **drawing_map()**, empleando el color y transparencia elegido con **drawing_color()** y/o el nivel de transparencia especificado en la última llamada a **drawing_alpha()**. Todos los puntos delimitados por esas cuatro coordenadas serán cambiados al color escogido.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el rectángulo (el escogido por la última llamada a **drawing_map()**). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X1 : Coordenada horizontal del primer punto
INT Y1 : Coordenada vertical del primer punto
INT X2 : Coordenada horizontal del segundo punto
INT Y2 : Coordenada vertical del segundo punto

**DRAW_FCIRCLE(X,Y,RADIO)**

Dibuja un círculo (con relleno de color) alrededor de un punto sobre el gráfico escogido en la última llamada a **drawing_map()**, empleando el color y transparencia elegido con **drawing_color()** y/o el nivel de transparencia especificado en la última llamada a **drawing_alpha()**

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el círculo (el escogido por la última llamada a **drawing_map()**). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X : Coordenada horizontal del centro
INT Y : Coordenada vertical del centro
INT RADIO : Radio en pixels

Vamos a ver un ejemplo sencillo sobre el uso de todas estas funciones nuevas.

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
Import "mod_screen";
Import "mod_rand";

Process Main()
begin
  set_mode(640,480,32);
  set_fps(1,1);
  drawing_map(0,0);
  repeat
     drawing_color(rgba(rand(0,255),rand(0,255),rand(0,255),rand(0,255)));
     draw_circle(rand(0,639),rand(0,200),rand(0,200));frame;
     draw_fcircle(rand(0,639),rand(0,200),rand(0,200)); frame;
     drawing_stipple(rand(1,4294967294));
     draw_line(rand(0,639),rand(0,479),rand(0,639),rand(0,479)); frame;
     draw_curve(rand(0,639),rand(0,479),rand(0,639),rand(0,479),rand(0,639),rand(0,479),r
and(0,639),rand(0,479),9);  frame;
     draw_box(rand(0,639),rand(0,479),rand(0,639),rand(0,479)); frame;
  until(key(_esc))
end
```

Otro ejemplo bastante parecido, donde unas cuantas primitivas cambian su color automáticamente, recorriendo todo el espectro de colores, desde el blanco (RGB=255,255,255) hasta el negro (RGB=0,0,0):

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
Import "mod_text";

Process Main()
private
  int r,g,b,color;
end
begin
  set_mode(320,240,32);
  write_var(0,0,0,0,r);
  write_var(0,0,10,0,g);
  write_var(0,0,20,0,b);
  drawing_map(0,0);
//Recorro todos los valores posibles de las distintas componentes RGB
  for(r=255;r>=0;r--)
          for(g=255;g>=0;g--)
                    for(b=255;b>=0;b--)
                       color=rgb(r,g,b);
                       drawing_color(color);
                       draw_line(50,50,270,50);
                       draw_rect(50,60,100,110);
                       draw_box(220,60,270,110);
                       draw_circle(75,130,15);
                       draw_fcircle(245,130,15);
                       frame;
                    end
           end
  end
end
```

Otro ejemplo ingenioso;el dibujo de una casa:

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
```

```
Process Main()
begin
  set_mode(320,240,32);
  drawing_map(0,0);
  drawing_color(rgb(255,255,0));
  draw_box(75,100,250,190); //paredes
  drawing_color(rgb(255,0,0));
  draw_rect(142,130,182,190); //puerta
  drawing_color(rgb(255,100,255));
  //Las tres siguientes líneas dibujan un triángulo (el techo)
  draw_line(75,100,162,25);
  draw_line(162,25,250,100);
  draw_line(250,100,75,100);
  drawing_color(rgb(0,0,255));
  draw_fcircle(162,70,20); //ventana
  drawing_color(rgb(100,0,0));
  draw_circle(172,160,5); //pomo
  while(!key(_esc)) frame; end
end
```

A continuación muestro otro ejemplo bastante interesante. Necesitarás un fichero FPG llamado "graficos.fpg" con dos imágenes. Una tendrá el código 001, será de 800x600 y estaría bien que tuviera el máximo de colores posible. Otra tendrá el código 002, será de 15x15 y representará la imagen del puntero del ratón (el tema de cómo introducir el uso del ratón en nuestros programas se explicará en profundidad más adelante en un capítulo posterior, no te preocupes ahora). Lo interesante de este ejemplo es ver lo que ocurre cuando movemos el cursor del ratón: pintamos permanentemente un rectángulo cuyo color de fondo es precisamente el color de la imagen de fondo sobre el que está situado el ratón. Para ello hemos utilizado la función ya vista map_get_pixel().

```
Import "mod_video";
Import "mod_draw";
Import "mod_screen";
Import "mod_key";
Import "mod_map";
Import "mod_mouse";

Process Main()
private
   int color;
   int fichero1;
end
begin
  set_mode(800,600,32);
  fichero1=load_fpg("graficos.fpg");
  put_screen(0, 1);
  mouse.graph=2;
  Loop
      If(key(_esc)) Break; End
      delete_draw(0);
      drawing_color(color);
      draw_box(320,180,380,220);
      // Cogemos el color del punto del fondo de la pantalla
      color = map_get_pixel(0,0,mouse.x, mouse.y);
      Frame;
  End
End
```

Con las funciones descritas hasta ahora, lo único que podemos hacer es dibujar una primitiva asociándolo a un gráfico (aunque sea el fondo). Si este gráfico dejara de visualizarse en pantalla, todas las primitivas pintadas en él desaparecerían con él. Para hacer que una primitiva tenga una entidad propia y no necesite de ningún gráfico extra para poder ser mostrada, lo primero que hay que hacer es "activar este modo de trabajo", con el cual lograremos tener un mayor control sobre una primitiva determinada (en concreto, podremos borrarla o moverla de forma independiente), ya que cada una de ellas dispondrá de un identificador propio y único, obtenido en la llamada a la función gráfica que la haya creado.

Para "activar este modo de trabajo", lo que hay que hacer es prescindir de **drawing_map()**. Y ya está. No

obstante, si se ha utilizado **drawing_map()** anteriormente y se quiere pasar de trabajar sobre mapas a dejar de hacerlo y usar este otro "modo de trabajo", entonces se deberá usar la función **drawing_z()**, la cual podemos considerar que "pasa" de un modo a otro. Esta función además tiene un parámetro que indica la Z a la que se verán las primitivas que se pinten a partir de entonces. ¿Y qué ganamos con esta manera de trabajar? Pues poder utilizar dos nuevas funciones: **delete_draw()** y **move_draw()** las cuales no podríamos usar si trabajáramos con **drawing_map()**, y que sirven, respectivamente, para borrar y mover en la pantalla una primitiva concreta.

**DRAWING_Z(Z)**

Escoge el nivel de profundidad a utilizar en las funciones de dibujo de primitivas, activando si no lo está ya "modo persistente de dibujado" , que independiza la primitiva de cualquier otro mapa.

Tal como he dicho, por defecto, las funciones de dibujo actúan sobre el fondo de pantalla o el gráfico especificado en la última llamada a la función **drawing_map()**. Sin embargo, una llamada a esta función activará un modo de funcionamiento alternativo, en el cual las primitivas gráficas (dibujadas con las funciones que empiezan por DRAW) persisten en pantalla como un gráfico independiente más, y podrán tener un identificador único.

Al igual que un proceso, estas primitivas gráficas se dibujan con una coordenada Z determinada, que permite que hayan procesos dibujados por encima o debajo de ellas. Esta función, además de activar este modo, permite escoger la coordenada Z que se aplicará a las primitivas gráficas dibujadas después de llamarla.

Para poder dibujar objetos con distinta coordenada Z, cada primitiva recuerda la coordenada Z elegida en el momento en que fue creada. Es posible, por lo tanto, hacer varias llamadas a **drawing_z()** entre llamadas a funciones de dibujo para poder cambiar el valor decoordenada Z con la que ir creando nuevas primitivas.

Las primitivas gráficas dibujadas de esta manera, tal como he dicho, podrán manipularse mediante las funciones **move_draw()** y **delete_draw()**.

Para desactivar este modo y volver a hacer dibujos directamente sobre el fondo de pantalla u otro gráfico, es preciso llamar a la función **drawing_map()** cuando se necesite.

PARÁMETROS : INT Z : Valor de profundidad, indicativo del orden de dibujo

**DELETE_DRAW(PRIMITIVA)**

Destruye (borra) una primitiva creada con una de las funciones de primitivas (utilizan todas el prefijo DRAW_), si ésta se ha creado después de una llamada a **drawing_z()**.

El parámetro que identifica a la primitiva es el valor devuelto por la función llamada para crearla.

Alternativamente, puede especificarse 0 como objeto, con lo que esta función destruirá todos los objetos creados con funciones de primitivas gráficas.

PARÁMETROS: INT OBJETO : Identificador de la primitiva a eliminar

**MOVE_DRAW(PRIMITIVA,X,Y)**

Mueve una primitiva gráfica creada con una de las funciones de primitivas (utilizantodas el prefijo DRAW_), si ésta se ha creado después de una llamada a DRAWING_Z, a una coordenada concreta de la pantalla.

Las coordenadas especificadas hacen referencia al primero de los puntos utilizados para dibujar una línea o curva, o bien al centro del círculo.

La gráfica no será deformada, sólo desplazada: en una línea, el segundo punto será desplazado en la misma magnitud que el primero.

PARÁMETROS:

INT OBJETO : Identificador del objeto a mover

| |
|---|
| INT X : Nueva coordenada horizontal<br>INT Y : Nueva coordenada vertical |

Para ver esto mejor, estudiemos unos ejemplos. Prueba de escribir esto:

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
Import "mod_screen";

Process Main()
private
  int a,b;
end
begin
  set_mode(640,480,32);
  a=10; b=10;
  drawing_map(0,0);
  drawing_color(rgb(0,255,0));
  repeat
    if(key(_up))b=b-1;end
    if(key(_down))b=b+1;end
    if(key(_left))a=a-1;end
    if(key(_right))a=a+1;end
    delete_draw(0);
    draw_line(0,0,a,b);
    frame;
   until (key(_esc))
 end
```

Si ejecutas el código anterior, podrás comprobar cómo con los cursores puedes cambiar la orientación de la línea verde, pero ¡no se borran las líneas pintadas en frames anteriores! Por lo que obtienes un barrido verde resultado de todos los dibujados de las líneas verdes anteriores, que no se han borrado. Y esto ocurre porque al especificar **drawing_map()**, estamos inutilizando la función **delete_draw()**, que recordemos que es junto con **move_draw()** una función que sólo actúa en el "modo persistente" de dibujado de primitivas. Puedes comprobar la diferencia de resultado si modificas el código anterior simplemente eliminando (o comentando) la línea *drawing_map(0,0);* Verás cómo ahora la línea verde cambia de orientación y se borra todo rastro de su posición en el anterior frame: esto es porque ahora sí que **delete_draw()** actúa. Fijarse por último que en este caso no hemos establecido ninguna Z para la línea: se dibuja con su valor por defecto.

Otro ejemplo:

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
Import "mod_rand";

Process Main()
private
  int idcirc;
end
begin
  set_mode(800,600,32);
  repeat
       drawing_color(rgba(112,164,195,227));
       idcirc=draw_circle(rand(0,639),rand(0,479),rand(0,320));
       frame;
       move_draw(idcirc,400,300);
       frame;
  until(key(_esc))
end
```

El ejemplo anterior lo que hace es pintar un círculo en una posición dada (primer frame) y después

moverlo para que su centro se situara en (400,300) (segundo frame). Y esto para cada una de las iteraciones del repeat/until. Si hubieras escrito por ejemplo antes del repeat una línea tal como ésta: *drawing_map(0,0);* el programa daría error, porque **move_draw()** no se puede utilizar con drawing_map(), recuerda.

Otro ejemplo interesante es el siguiente, sobre el uso de los identificadores de primitivas: se dibujan tres círculos pero sólo uno se genera con identificador, el cual usamos para borrarlo unívoca y específicamente pulsando una tecla.

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";

Process Main()
private
  int idcircle;
end
begin
  set_mode(640,480,32);
  drawing_color(rgba(255,0,0,255));
  idcircle=draw_circle(100,100,100);
  drawing_color(rgba(0,255,0,255));
  draw_circle(200,200,100);
  drawing_color(rgba(0,0,255,255));
  draw_circle(300,300,100);
  repeat
       if(key(_a)) delete_draw(idcircle); end
       frame;
  until(key(_esc))
end
```

Y finalmente, otro ejemplo sobre el uso de la Z en el dibujado de primitivas. En él, primero se establece como profundidad común a todas las primitivas Z=1, y a continuación se pintan tres círculos. Fijarse que aunque la profundidad sea la misma para todos, el orden de dibujado impone una jerarquía: la primera primitiva en pintarse aparecerá por encima de la segunda, y ésta sobre la tercera: por eso el círculo rojo se ve por encima del verde y el azul por debajo de ambos. A partir de ahí, cada vez que se pulsa con la tecla "s", se disminuye la profundidad de dibujado, borramos (sólo) el círculo azul y volvemos a repintarlo, momento en el que se utiliza la nueva profundidad y se muestra (si es el caso) algún cambio en pantalla. Lo mismo ocurre si se pulsa la tecla "a", pero aumentando la profundidad en vez de disminuirla. Lo que se puede comprobar utilizando dichas teclas es que cuando la nueva profundidad es menor que la inicial con la que se pintaron los círculos rojo y verde, el nuevo círculo azul se pinta por encima de ellos, y cuando la nueva profundidad aumenta, sigue estando por debajo (de hecho, "más" por debajo).

```
Import "mod_video";
Import "mod_draw";
Import "mod_key";
Import "mod_map";
Import "mod_text";

Process Main()
private
  int idcircle;
  int zeta;
end
begin
  set_mode(640,480,32);
  write_var(0,0,0,0,zeta);
  drawing_z(1);
  drawing_color(rgba(255,0,0,255));
  idcircle=draw_fcircle(300,200,100);
  drawing_color(rgba(0,255,0,255));
  draw_fcircle(400,300,75);
  drawing_color(rgba(0,0,255,255));
  idcircle=draw_fcircle(400,200,50);
  repeat
       if(key(_a))
                      zeta++;
```

```
                    drawing_z(zeta);
                    delete_draw(idcircle);
                idcircle=draw_fcircle(400,200,50);
        end
        if(key(_s))
                    zeta--;
                    drawing_z(zeta);
                    delete_draw(idcircle);
                idcircle=draw_fcircle(400,200,50);
        end
        frame;
  until(key(_esc))
end
```

## *Crear, editar y grabar FPGs dinámicamente desde código Bennu

En este apartado voy a intentar explicaros el funcionamiento de un puñado de funciones de Bennu (todas definidas en el módulo "mod_map") que nos permitirán, entre otras cosas, poder programar un rudimentario "FPGEdit" que lo único que haga de momento sea crear un archivo FPG, añadir en él imágenes (o eliminarlas) y grabarlo en disco. Este ejercicio nos será muy útil para no depender de editores externos, los cuales puede que sólo funcionen en una plataforma, o no nos agrade su funcionamiento: con las funciones siguientes nos podremos crear nuestro propio editor FPG a nuestra medida.

### NEW_FPG()

Esta función crea en memoria una nueva librería FPG y devuelve su código numérico de librería. A partir de entonces, es posible añadir nuevos gráficos a esta librería mediante la función **fpg_add()** , y grabarla posteriormente a disco con **save_fpg()**

VALOR DE RETORNO: INT : Código de la nueva librería FPG

### FPG_ADD ( LIBRERÍA, GRÁFICO, LIBRERÍA-ORIGEN, GRÁFICO-ORIGEN )

Esta función permite modificar una librería FPG en memoria, añadiendo un nuevo gráfico a partir de otro gráfico ya cargado previamente en memoria.

La librería de destino debe ser el código de una librería FPG obtenido mediante las funciones **load_fpg()** ó **new_fpg()**. La librería de origen debe ser el código de otra librería en memoria, aunque también puede identificar la propia librería de destino (para hacer copias de gráficos dentro de la misma) o bien la librería 0, para poder especificar gráficos obtenidos con funciones como **new_map()**

La función **fpg_add()** crea una copia del gráfico original, el cual seguirá válido y accesible.

Si ya hubiese un gráfico con el mismo código en la librería de destino, éste será eliminado de memoria y sustituido por el nuevo.

La mayor utilidad de esta función es crear nuevas librerías de gráficos para guardarlas luego con **save_fpg()**.

Todos los gráficos añadidos a una misma librería deben tener la misma profundidad de color (1, 8, 16 ó 32 bits). Añadir un gráfico a una librería cuando ya hay en la misma algún gráfico de otro tipo se considera una condición de error.

PARÁMETROS:

INT LIBRERÍA : Código de la librería a modificar
INT GRÁFICO : Código deseado para el nuevo gráfico, de 0 a 999
INT LIBRERÍA-ORIGEN : Código de la librería origen
INTGRÁFICO-ORIGEN : Código del gráfico original

### SAVE_FPG ( LIBRERÍA, "FICHERO" )

Esta función crea o sobreescribe un fichero en disco con el contenido de una librería FPG en memoria que puede haberse creado con la función **new_fpg()**, o recuperado de disco mediante **load_fpg()**. Los cambios realizados en

memoria con funciones como **fpg_add()** ó **unload_map()** no afectan al fichero original en disco, así que es preciso utilizar esta función para sobreescribirlo si así se desea.

PARÁMETROS:

INT LIBRERÍA : Código de la librería a guardar
STRING FICHERO : Nombre del fichero

**UNLOAD_MAP(LIBRERÍA, GRÁFICO)**

Ya sabemos que esta función libera la memoria ocupada por un gráfico que puede formar parte de una librería FPG, o bien haber sido creado independientemente por **new_map()** , o bien haber sido recuperado de disco por una función como **load_png()**. Pero ¿por qué la incluyo en esta lista de funciones relacionadas con los FPGs? Porque es posible utilizar esta función para quitar gráficos de un fichero FPG (lo contrario de **fpg_add()**). Los cambios realizados por esta función sólo afectan al FPG en memoria: para actualizar el fichero FPG con ellos recuerda que es preciso usar la función **save_fpg()** .

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico dentro de la librería

**FPG_DEL(LIBRERIA)**

Esta función elimina de la memoria una librería FPG creada anteriormente allí con **new_fpg().** No se puede usar para descargar librerías cargadas con **load_fpg()**, para ello existe la función **unload_fpg().** La utilidad de esta función radica sobretodo en la eliminación de un FPG de la memoria después de haber sido almacenado en disco mediante **save_fpg()**.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

**MAP_SET_NAME ( LIBRERÍA, GRÁFICO, "NOMBRE" )**

Esta función cambia el nombre de un gráfico en memoria.Todo gráfico tiene un nombre de hasta 32 caracteres, que puede cambiarse mediante esta función, o recuperarse mediante **map_name()**. Este nombre es especialmente útil cuando el gráfico forma parte de una librería FPG, ya que de otra manera no habría otro sistema aparte del número del gráfico para distinguir uno de otro.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico dentro de la librería
STRING NOMBRE : Nuevo nombre del gráfico

**MAP_NAME ( LIBRERÍA, GRÁFICO )**

Devuelve el nombre de un gráfico.Todo gráfico tiene un nombre de hasta 32 caracteres, que puede recuperarse mediante esta función, o cambiarse mediante **map_set_name()**. Este nombre es especialmente útil cuando el gráfico forma parte de una librería FPG, ya que de otra manera no habría otro sistema aparte del número del gráfico para distinguir uno de otro.

PARÁMETROS:

INT LIBRERIA: Número de librería FPG
INT GRAFICO: Número de gráfico dentro de la librería

VALOR DEVUELTO: STRING : Nombre del gráfico

Vamos a empezar con un ejemplo que es una prueba de concepto: una aplicación que cargará un archivo FPG previamente existente (llamado “antiguo.fpg”) con dos imágenes de códigos 001 y 002, y guardará las mismas

imágenes en un nuevo archivo FPG llamado “nuevo.fpg”. Esto sería así:

```
Import "mod_map";

process main()
private
           int idfpg1;
           int idfpg2;
end
begin
           idfpg1=load_fpg("antiguo.fpg");
           idfpg2=new_fpg();
/*Los gráficos se introducen en el nuevo archivo FPG en el mismo orden que estaban en el antiguo*/
           fpg_add(idfpg2,1,idfpg1,1);
           fpg_add(idfpg2,2,idfpg1,2);
           save_fpg(idfpg2,"nuevo.fpg");
end
```

Puedes comprobar cómo el nuevo.fpg se puede abrir perfectamente con cualquiera de las herramientas que disponemos para ello (el "FPGEdit", por ejemplo) y que contiene exactamente las mismas imágenes que "antiguo.fpg".

Otro ejemplo sencillo: el siguiente código genera un gráfico en memoria mediante primitivas gráficas y lo incluye en un FPG creado al efecto. Así pues, como resultado de la ejecución del programa obtendremos un fichero FPG con un gráfico en su interior, todo generado por código.

```
import "mod_screen";
import "mod_map";
import "mod_draw";

process main()
private
 int cubo;
 int lib;
end
begin
  cubo=new_map(100,100,32);
  drawing_map(0,cubo);
  drawing_color(rgb(0,0,200));
  draw_box(50,0,100,100);
  drawing_color(rgb(0,100,200));
  draw_box(0,0,50,100);
  drawing_color(rgba(100,0,200,50));
  draw_box(25,0,75,25);
  /* Hole */
  drawing_color(rgba(255,255,0,200));
  draw_box(35,35,65,65);
  drawing_color(rgba(255,0,255,64));
  draw_box(45,45,55,55);

  lib = new_fpg();
  fpg_add(lib,1,0,cubo);
  save_fpg(lib,"fpg32.fpg");
End
```

A partir de aquí podríamos programarnos nosotros mismos pequeñas aplicaciones que manipulen archivos FPG a nuestro antojo, más útiles y prácticas en nuestro día a día. Por ejemplo, podríamos programar un creador de FPGs a partir de múltiples imágenes PNG que tengamos grabadas en una carpeta determinada. Como muestra, el siguiente ejemplo, el cual genera un FPG llamado (siempre) "salida.fpg":

```
import "mod_map";
import "mod_say";
import "mod_string";
import "mod_file";

private
```

```
  int i;
  int f;
end
Begin
    f=new_fpg();
    from i=1 to 999;
        If(file_exists(substr("000"+itoa(i),-3)+".png"))
            fpg_add(f,i,0,load_png(substr("000"+itoa(i),-3)+".png"));
        end
    End
    save_fpg(f,"salida.fpg");
    fpg_del(f);
End
```

El código anterior funciona perfectamente pero hay que tener en cuenta una serie de requisitos:

1.-Todas las imágenes a incluir en el FPG deberán estar situadas en una misma carpeta. Dicha carpeta, además, deberá incluir también el propio código anterior, ya que éste va a buscar las imágenes en la misma carpeta donde está él situado.

2.-Todas las imágenes PNG deberán llamarse obligatoriamente respetando el patrón siguiente: su nombre será un número siempre de tres cifras -completando con 0 si es necesario- (por ejemplo: “004”, “065”, “983”) y su extensión será siempre “png” en minúsculas. Ese número de tres cifras será el identificador de la imagen dentro del FPG. Si el nombre de la imagen no cumple este requisito, no será detectado y no se incluirá en el FPG

Estos requisitos son perfectamente subsanables con un poco más de programación, pero teniéndolos en cuenta, este código tan sencillo es perfectamente funcional. Lo que sí se podría optimizar más en este programa es el bucle from, ya que se realizan siempre 999 iteraciones (el número máximo de imágenes que pueden ser albergadas en un mismo FPG) independientemente de si existen 1, 2 ó 999 imágenes para incluir en él. Debería haber un sistema para detectar que no hay más imágenes que incluir y parar en consecuencia el bucle.

Como habrás podido observar, en este código se han introducido unas cuantas funciones que no conocemos todavía relacionadas con el manejo de ficheros en disco y cadenas (**file_exists(), substr(), itoa()**). Ya las estudiaremos más a fondo más adelante, simplemente sirven en conjunto para detectar que efectivamente existan ficheros llamados de la forma XXX.png (donde X es un dígito cualquiera) en la carpeta actual.

Otro ejemplo de generador de FPG algo más sofisticado es el siguiente:

```
Import "mod_map";
Import "mod_text";
Import "mod_key";
Import "mod_string";
Import "mod_file";
Import "mod_video";
Import "mod_proc";

process main()
Private
        int i;
        string entry="";
        int idfpg;
        int idgraf;
        int porcentaje;
end
Begin
/*La orden set_mode es opcional: por defecto si no se establece la profundidad de color,
será 32, y por lo tanto, el FPG creado será de esta profundidad. Sólo sería importante
establecerla si se desea crear fpgs de otras profundidades (16,8 bits).*/
        set_mode(300,25,32);
        set_fps(0,0); //Ajusto la velocidad del programa a la máxima posible
        write(0,0,0,0,"Nombre del FPG a crear (sin la extensión):");

//Loop de introducción por teclado del nombre solicitado
        write_var(0,0,10,0,entry);//Para que se vea lo que se escribe mientras se escribe
        Loop
/*En las líneas siguiente utilizaremos una variable global predefinida todavía no
```

```
explicada: SCAN_CODE.No te preocupes ahora de ella, no es importante. Sólo has de saber
que actúa de forma similar -aunque no idéntica- a la función key(): es decir, sirve para
detectar qué tecla se ha pulsado. Sus valores pueden ser las mismas constantes que se
usan en la función key()*/
            If(scan_code==_backspace)entry = substr (entry, 0, -1); End //Borro una letra
           If(scan_code==_esc ) exit(); end //Me arrepiento y quiero salir del programa
            If(scan_code==_enter) Break ; End //Dejo de introducir nada más
            scan_code=0; //El programa cree que no hay pulsada ninguna tecla
/*Si se ha pulsado alguna caracter imprimible (valores ASCII > 32) esta línea sirve para
recoger ese caracter y lo almacena en la variable de tipo cadena "entry".*/
            If (ascii >= 32) entry = entry +chr(ascii) ; ascii =  0 ;  End
            Frame;
        End

//Proceso de introducción de los gráficos en el FPG
        idfpg=new_fpg();
        delete_text(0);
        write(0,0,0,0,"Espere...Porcentaje de completado:");
        write_var(0,220,0,0,porcentaje);
        for(i=1;i<999;i++)
                    porcentaje=i*100/999;
                    If(file_exists(itoa(i)+".png"))
                              idgraf=load_png(itoa(i)+".png");
                              fpg_add(idfpg,i,0,idgraf);
                    Else
/*/Se supone que si no se encuentra un fichero, ya no habrá más después, pero no se puede
asegurar.Por eso comento el break y simplemente muestro un mensaje indicando el hecho.*/
                              // break;
                             delete_text(3);
                             write(0,0,10,0,"No se encontró el gráfico número: ");
                             write_var(0,220,10,0,i);
                    End
                    Frame;
         End
        save_fpg(idfpg,entry + ".fpg");
        delete_text(all_text);
        write(0,0,0,0,"Se ha finalizado la introducción.");
        write(0,0,10,0,"Pulse ENTER para finalizar el programa.");
        while(!key(_enter)) Frame; End
End
```

Este código es más sofisticado que el anterior en el sentido de que deja elegir al usuario el nombre del fichero FPG a generar, y muestra visualmente el avance del proceso de introducción de imágenes en el FPG (aunque se siguen haciendo 999 iteraciones). No obstante, sigue obligando a cumplir unos requisitos:

1.-Todas las imágenes a incluir en el FPG deberán estar situadas en una misma carpeta. Dicha carpeta, además, deberá incluir también el propio código anterior, ya que éste va a buscar las imágenes en la misma carpeta donde está él situado.

2.-Todas las imágenes PNG deberán llamarse obligatoriamente respetando el patrón siguiente: su nombre será un número de cualquier número de cifras (por ejemplo: "4", "65", "983") y su extensión será siempre "png" en minúsculas. Ese número será el identificador de la imagen dentro del FPG. Si el nombre de la imagen no cumple este requisito, no será detectado y no se incluirá en el FPG

En vez de que preguntara el nombre deseado del FPG una vez iniciado el programa, en Bennu existe otra posibilidad de indicar ese nombre: escribirlo en el intérprete de comandos como parámetro de nuestro programa en el mismo momento de ejecutarlo. Pero esta posibilidad no hemos visto todavía cómo se implementa: la estudiaremos más adelante.

Otro tipo de programa muy útil que podemos desarrollar es un extractor de imágenes PNG de un FPG dado. El siguiente ejemplo extrae de un FPG (se pregunta al usuario cuál en concreto) todas sus imágenes PNG en una carpeta dentro del directorio actual, desde donde se ejecuta el programa, llamada igual que el archivo FPG, y les pone de nombre en el fichero el número identificador que tenían dentro del FPG (es decir: 1.png, 2.png,etc,hasta 999.png).

```
Import "mod_video";
Import "mod_map";
```

```
Import "mod_text";
Import "mod_key";
Import "mod_string";
Import "mod_proc";
Import "mod_file";
Import "mod_dir";

process main()
private
        int i=1;
        String entry;
        int idfpg;
        int idgraf;
        int porcentaje;
end
Begin
/*La orden set_mode es opcional: por defecto si no se establece la profundidad de color,
será 32, y por lo tanto, el FPG con el que se trabajará tendrá que ser obligatoriamente
de esta profundidad. Sólo sería importante establecerla si se desea extraer imágenes de
fpgs de otras profundidades (16,8 bits).*/
          set_mode(300,25,32);
          set_fps(0,0);
//Mientras no se introduzca un nombre de FPG válido, no se saldrá de aquí
          While(idfpg<=0)
                    delete_text(all_text);
                    entry = "" ;
                    write(0,0,0,0,"Nombre del FPG a tratar (sin la extensión):");
//Mismo loop de introducción por teclado del nombre solicitado que el ejemplo anterior
                    write_var(0,0,10,0,entry);
                    Loop
                        If(scan_code == _backspace) entry = substr (entry, 0, -1); End
                        If(scan_code == _esc ) exit(); end
                        If(scan_code == _enter) Break ; End
                        scan_code = 0 ;
                        If (ascii >= 32) entry = entry + chr(ascii);ascii =  0 ; End
                        Frame ;
                    End
                    idfpg=file_exists(entry + ".fpg");
                    frame;
          End
          idfpg=load_fpg(entry + ".fpg");
          delete_text(0);
          write(0,0,0,0,"Espere...Porcentaje de completado:");
          write_var(0,220,0,0,porcentaje);
          mkdir(entry);
          for(i=1;i<999;i++)
                    porcentaje=i*100/999;
          If(map_info(idfpg,i,g_width)!=0 AND map_info(idfpg,i,g_height)!=0)
                              save_png(idfpg,i,entry+"/"+itoa(i)+".png");
/*Si se cumple el else, ya hemos llegado al final de la lista de todos los gráficos y no
tiene sentido continuar*/
                    Else
                              break;
                    End
                    Frame;
          End
          delete_text(all_text);
          write(0,0,0,0,"Se ha finalizado la extracción.");
          write(0,0,10,0,"Pulse ENTER para finalizar el programa.");
          While(!key(_enter)) Frame; End
End
```

El requisito que ha de cumplir el código anterior es que el FPG a extraer ha de estar en la misma carpeta que aquél.

Con los códigos anteriores obtenemos una parte pequeña de la funcionalidad que nos ofrece el “FPGEdit” para archivos FPG, porque, no obstante, falta bastante todavía. Por ejemplo, se podría incorporar alguna función que

nos permitiera eliminar imágenes del FPG y seguramente también otra que las listara, entre otras ideas. En el último capítulo de este texto se presentan dos ejemplos más de códigos gestores de FPGs más completos (y complicados) para poder ser estudiados con más detenimiento por quien quiera que tenga un conocimiento relativamente avanzado de Bennu.

Finamente, y por otra parte, no quisiera dejar de comentar al final de este apartado dedicado a los contenedores FPG una posibilidad curiosa que tenemos disponible a la hora de cargar ficheros FPG. Todo el mundo sabe a estas alturas que para cargar estos contenedores hemos de escribir en nuestro código una línea semejante a

```
idfpg=load_fpg("ficherofpg.fpg");
```

Pero existe una alternativa para hacer lo mismo, que es escribiendo esto:

```
load_fpg("ficherofpg.fpg", &idfpg);
```

Fijarse que ahora el identificador del FPG cargado se escribe como segundo parámetro (en vez de ser devuelto), y además, se le añade al principio el símbolo "&" (en próximos capítulos veremos qué significa).Pero, ¿hay alguna diferencia entre esta nueva forma de hacer la carga y la que ya sabíamos? Pues sí, hay una diferencia sutil: esta segunda forma tiene su utilidad básicamente cuando se cargan ficheros FPG muy grandes (con muchísimos gráficos en su interior o con gráficos de dimensiones muy elevadas). Algunos ficheros FPG pueden ser tan grandes que pueden tardar demasiado tiempo en acabar de cargarse todo completo, y como hasta que no se acaba ese proceso el programa se queda detenido en ese línea y no continúa, el juego se quedaría "congelado" durante unos instantes. Con esta segunda manera de cargar FPGs se evita este efecto, porque los diferentes gráficos del interior del contenedor se irán cargando "en segundo plano", continuando la ejecución del programa mientras tanto. Esto es muy útil, ya hemos dicho, si el archivo es tan grande que se corre el riesgo de parecer que el programa se haya bloqueado, o si quieres ir haciendo una animación mientras se acaba de cargar entero.

# CAPITULO 4

## Procesos

### *Concepto de proceso

Se llama proceso a cada uno de los diferentes objetos que hay dentro de un juego. Por ejemplo, en un juego de marcianos un proceso será la nave que maneja el jugador, otro proceso será cada enemigo, y cada explosión, y cada disparo,etc. Normalmente se equiparan a los diferentes gráficos en movimiento (los llamados sprites), aunque pueden existir procesos que no aparezcan en pantalla.

Los procesos son "partes" de código. Cada proceso tiene su propio código que se ejecuta y se relaciona con los demás procesos según sus órdenes concretas; puede constar de un bucle (infinito o no), o bien ejecutarse sólo una vez hasta que llega a su fin y se termina. A diferencia de otros lenguajes, Bennu se basa en programación concurrente; esto quiere decir que en lugar de seguir línea por línea el programa en un orden, éste se ejecuta en trozos que avanzan al mismo tiempo: los procesos. Esto al principio resulta confuso, pero una vez te acostumbras, ves que gracias a ello cada proceso actúa casi de forma independiente y es muy útil por ejemplo en juegos como donde cada personaje es un proceso independiente y así no tiene que esperar a que se ejecuten antes los demás.

De esta manera, podemos crear programas más flexibles muy fácilmente. Además, la detección de errores es también más sencilla, porque si todo funciona excepto el movimiento de tu nave, sabrás exactamente dónde tienes que buscar. Pero esta manera de codificar también exige una planificación previa con lápiz y papel y decidir antes de ponerse a teclear código como un loco cuántos procesos se van a crear, de qué tipo, etc, para tenerlo claro antes de perderse.No es broma: es muy importante no llegar al momento donde están en mitad del código pensando por qué no funciona sin tener ni idea.

Veamos todo esto con un ejemplo. Ya sabemos que el ordenador ejecuta los programas línea a línea, desde arriba hasta el final, parecido a leer un libro. Si entonces hacemos un videojuego de matar enemigos, por poner un caso, podríamos escribir algo así como:

*Principio:*
*Dibujar tu nave*
*Dibujar los enemigos*
*Dibujar los disparos*
*Chequear si hay colisiones*
*Si un disparo toca un enemigo, matarlo*
*Si un del enemigo te toca: matarte*
*Chequear si hay algún movimiento de joystick*
*Calcular la nueva posición acorde al movimiento del joystick*
*Crear un nuevo disparo si has apretado el botón de disparar*
*Calcular los movimientos de los enemigos*
*Ir al Principio.*

El ejemplo sólo se ha escrito para mostrarte la manera de programar que en Bennu NO se utiliza. Aunque éste haga todo lo necesario (evidentemente, se necesitan más partes del programa, como concretar qué es lo que pasa cuando un enemigo o tú muere, etc, pero esto no es importante ahora), lo hace siempre desde arriba hasta abajo y volviendo a empezar, para hacer siempre lo mismo una y otra vez. Y Bennu no funciona así. ¿Por qué? Porque cuando el videojuego se vuelve cada vez más complicado,el bucle del programa se hace cada vez mayor y el código se vuelve pronto desordenado y caótico. La solución, ya digo, pasa por la multitarea (también llamada programación concurrente). Multitarea significa que más de un programa (es decir, los procesos) están ejecutándose al mismo tiempo. Así, en vez de un único bucle inmenso como el anterior, donde las líneas una tras otra se ejecutan para hacer todos los movimientos, colisiones, chequeos,etc, se tiene más programas/procesos pequeños que hacen todo esto continuamente por separado. Entonces,básicamente tendríamos los siguientes procesos funcionando al mismo tiempo para lograr lo

mismo que con el ejemplo anterior:

*TuNave (movimiento,dibujo,proceso_creación_disparos, chequeo_colisiones)*
*Enemigo (movimiento, dibujo, chequeo_colisiones)*
*Disparo (movimiento, dibujo, chequeo_colisiones)*

En vez de tener un código grande y desordenado, se tiene tres códigos simples funcionando a la vez.

Otro detalle a tener en cuenta es que es posible ejecutar múltiples instancias de un proceso. Es decir, si cada enemigo y disparo es un proceso, podremos ejecutar ese proceso más de una vez y simultáneamente, por lo que tendríamos tantos enemigos o disparos en pantalla como procesos de ese tipo estuvieran funcionando a la vez, de forma independiente cada uno de ellos.

## *Las variables locales predefinidas GRAPH, FILE, X e Y

Hasta ahora sólo hemos utilizado un tipo de variables, las variables "privadas" (independientemente de si son INT, STRING o lo que sea). En Bennu existen cuatro tipos generales de variables: "privadas", "públicas","globales" y "locales", y cada uno de estos tipos existe porque está especialmente pensado para determinados usos específicos en los que un tipo es mejor que los demás. No obstante, de cada uno de ellos hablaremos más adelante: por ahora nos va a ser irrelevante si son de uno o de otro. Lo único que debemos saber es que existen entros cuatro tipos, y de momento ya está.

Hasta ahora hemos visto también que para poder utilizar una variable cualquiera (sólo hemos usado variables privadas, pero para las públicas, locales y las globales pasaría lo mismo) la tenemos previamente que declarar, porque si no el programa no puede reconocer dicha variable en el código. Esto es así para todas las variables que nosotros creemos.

No obstante, Bennu dispone de un conjunto de variables (que son o globales o locales, pero nunca privadas o públicas) que existen y que pueden ser usadas en nuestro código sin ninguna necesidad de declararlas. Son variables listas para usar, predefinidas con un tipo de datos concreto –normalmente INT-, y están disponibles en cualquier programa Bennu. Evidentemente, nosotros no podemos crear variables que tengan el mismo nombre que alguna de las predefinidas existentes, porque si no habría confusiones en los nombres. Ya vimos en un capítulo anterior unos ejemplos de variables predefinidas (concretemente, globales):***TEXT_Z*,*FPS*,*FRAME_TIME*,*SPEED_GAUGE*...**

¿Por qué existen estas variables predefinidas? Porque cada una de ellas realiza una tarea muy específica y especialmente encomendada a ella. Es decir, que estas variables predefinidas cada una de ella existe por una razón determinada; y se tendrán que usar en consecuencia. No son variables generales como las que podamos crear nosotros, en las que diciendo que son de un tipo de datos determinado pueden almacenar cualquier cosa. Cada variable predefinida tiene un significado concreto, y los valores que pueda almacenar en consecuencia serán unos determinados. Las variables predefinidas se utilizan, básicamente, para controlar los diferentes dispositivos del ordenador (ratón,teclado, pantalla, tarjeta de sonido,joystick…)y los gráficos de los juegos.

A continuación vamos a conocer el significado de dos variables predefinidas (locales) enteras muy importantes, como son ***GRAPH*** y ***FILE***. Antes de nada, no obstante, lo primero que deberemos de tener en cuenta a partir de ahora es que para poder utilizarlas, deberemos importar siempre el módulo "mod_video".

Como primera aproximación, ya hemos dicho que podemos asumir que en un juego de matar enemigos, por ejemplo, cada enemigo representará un proceso diferente, los disparos otro, nuestra nave otro, etc. Lo primero que tendríamos que hacer para que todos estos procesos se vieran en pantalla es asociarles a cada uno de ellos una imagen: los enemigos tienen una imagen (o varias si queremos hacer enemigos diferentes), los disparos también, nuestra nave también,etc. De hecho, hasta ahora las únicas imágenes que hemos sabido poner en pantalla son las imágenes de fondo, pero cada proceso deberá tener su propia imagen (o imágenes) si queremos que se visualice de forma independiente. Aquí es donde interviene la variable ***GRAPH***.

Cuando creamos un proceso, la manera de asignarle una imagen es estableciendo un valor para la variable ***GRAPH*** dentro de dicho proceso. Cada proceso puede tener asignado un valor diferente a ***GRAPH*** de forma independiente: los distintos valores no se interferirán unos con otros porque el código de cada proceso se ejecuta aislado

del de los demás, y la variable ***GRAPH*** guarda la asociación de los distintos valores que puede tener en los diferentes procesos con el proceso al que le corresponde (de hecho, esto que acabo de decir es la definición de variable local, que ya la veremos más adelante). El valor que puede tener esta variable ***GRAPH*** es un número entero, el cual será el identificador de la imagen que se asociará al proceso y lo representará visualmente, identificador que podrá haber sido devuelto por **load_png()**, ó **new_map()** ó **write_in_map()** ó similar.

Si en vez de utilizar **load_png()** -o las funciones de la familia "map"- utilizamos **load_fpg()**, aparte de ***GRAPH*** necesitaríamos otra variable local predefinida entera: ***FILE***. Esta variable toma como valor el identificador del archivo FPG cargado mediante **load_fpg()**, de tal manera que entonces ***GRAPH*** pasa a valor el código numérico interno que tiene la imagen buscada dentro del archivo FPG. Así pues, para definir la imagen de un proceso que ha sido cargada con **load_fpg()**, tendríamos que indicarle en qué archivo FPG se encuentra, pasándole el identificador de ese archivo a la variable ***FILE***, y dentro del FPG, de qué imagen se trata, pasándole el código numérico interno a la variable ***GRAPH***.

Además de ***GRAPH*** y ***FILE***, existen otras dos variables locales predefinidas más de las que querría decir cuatro cosas ahora: son la variable ***X*** y la variable ***Y***. Éstas tienen como valor la coordenada x e y respectivamente del centro de la imagen dada por ***GRAPH*** (y ***FILE*** en su caso). Es decir, sirven para establecer las coordenadas del centro de la imagen asociada al proceso donde se definen. Igual que ***GRAPH***, cada proceso puede dar valores diferentes a ***X*** e ***Y*** de forma independiente unos de otros porque cada proceso no interfiere – a priori- en los demás: cada proceso tiene una copia particular de las variables ***X*** e ***Y*** con sus valores propios.

Pronto veremos ejemplos de todo esto, donde se verá mucho más claro lo explicado arriba.

## *Creación de un proceso

Cada proceso debe empezar con la palabra reservada PROCESS, el nombre del proceso y la lista entre paréntesis de parámetros –especificando su tipo- que puede tener, si es que tiene. Los parámetros son una lista de variables en los que el proceso va a recibir diferentes valores cada vez que sea invocado (es decir, llamado o utilizado desde otro proceso), ya los estudiaremos más adelante. Los paréntesis son obligatorios incluso cuando el proceso no tenga parámetros. Seguidamente viene la declaración de variables que necesite,las cuales sólo podrán ser (en un principio) de tipo "privadas" –las que venimos usando hasta ahora-, y luego un BEGIN, el código del proceso y un END. Es decir:

```
PROCESS NombreProceso ( tipo param1, tipo param2,…)
PRIVATE
        Declaraciones de variables privadas;
END
BEGIN
        Código del proceso;
END
```

Fíjate que la estructura de un proceso se parece mucho a lo que hemos visto hasta ahora, que es la estructura del programa principal. De hecho, el programa principal no es más que otro proceso más, un proceso que se llama obligatoriamente siempre igual: "Main()".

Ese proceso puede ejecutar todas las órdenes que se quieran, llamar a otros procesos, "matarlos" (hacer que dejen de ejecutar su código), congelarlos (hacer que paren de ejecutar su código, pero sin "matarlo"), dormirlos (congelarlo haciendo que su gráfico no aparezca en pantalla), interactuar con ellos y cualquier otra cosa que se nos ocurra. Como hemos dicho, se puede hacer más de una llamada a un proceso concreto (ej.: cada vez que el protagonista tenga que disparar se llama al mismo proceso disparo, ahorrando muchísimas líneas de código). Y pueden existir infinidad de procesos a la vez, cada uno ejecutando su código único, hasta que la memoria de la máquina lo permita (hay que tener en cuenta que si existen muchos procesos el ordenador se puede ralentizar).

Es importante diferenciar entre los bloques PROCESS de los programas, que definen el comportamiento de un tipo concreto de proceso, y lo que es un proceso concreto del juego en tiempo de ejecución (lo que se llama técnicamente "instancia" de un proceso). La instancia de un proceso es un objeto individual del juego cuyo

comportamiento está regido por uno de los bloques PROCESS del programa: si tenemos un proceso “disparo”, podrá haber múltiples instancias (es decir, múltiples disparos) en nuestro juego, cada uno de ellos ejecutando a la vez el mismo código del bloque PROCESS “disparo”.

El concepto de procesos parece claro ya, pero: ¿cómo insertar el código de los procesos dentro del programa principal (es decir, dentro del proceso “Main()”)?.¿Y dónde aparecen esas variables GRAPH y FILE comentadas antes?
He aquí un ejemplo que lo explica. Es vital entenderlo para poder continuar, así que atento. Necesitarás un archivo FPG llamado “graficos.fpg” con una imagen –de 30x30 estaría bien- en su interior de código 1:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";

GLOBAL
    INT id1;
END

PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
END

PROCESS personaje()
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        IF (key(_down)) y=y+10; END
        IF (key(_left)) x=x-10; END
        IF (key(_right)) x=x+10; END
        IF (key(_esc)) break; END
        FRAME;
    END
END
```

Fíjate que el código se estructura en dos partes, que son los dos procesos de los que consta el programa. La primera parte es la que ya conocemos, que es el proceso correspondiente al programa principal, el proceso “Main()”, y la segunda a un proceso llamado “personaje()” que hemos creado. Si hubiera más procesos, la estructura se repetiría con los correspondientes bloques PROCESS/END para cada proceso extra. Antes de nada, conviene recordar que, aunque a partir de ahora nos encontremos con códigos con múltiples procesos, Bennu comenzará a ejecutar nuestro programa siempre por la misma puerta de entrada: el proceso “Main()”. Éste es el proceso principal (el único imprescindible, de hecho) que será siempre el primero en comenzar a ejecutarse, y a partir del cual se irán lanzando todos los demás procesos.

Lo primero que salta a la vista del código anterior es que, a diferencia de hasta ahora, hemos declarado una variable en el programa principal de tipo “global”. Y además, el bloque de declaración “global/end” se escribe fuera de cualquier proceso, justo después de las líneas “import” (no como el bloque “private/end”, que se escribe dentro del proceso que corresponda). El usar este tipo de variable es necesario para que el ejemplo funcione, (si fuera privada no lo haría), pero el por qué lo explicaré dentro de poco, ahora no es relevante.

Seguidamente puedes observar ponemos la resolución y la profundidad del juego, cargamos un archivo FPG al cual le damos el identificador “id1”, y finalmente, y aquí está la novedad, llamamos –o mejor dicho, creamos- al proceso “personaje()”.

Fíjate que la manera de crear un proceso es simplemente escribir su nombre, y si tuviera parámetros, poner los valores que queramos que tengan a la hora de la creación del proceso. En este caso, “personaje()” es un proceso sin parámetros, y por tanto, sólo se ponen los paréntesis vacíos. Al crearse este proceso, si éste es visible porque está asociado a un gráfico, se verá ese gráfico por pantalla. Si se volviera a ejecutar la línea *personaje();* otra vez (porque estuviera dentro de un bucle, por ejemplo), se crearía otro proceso idéntico en el juego y por tanto tendríamos dos procesos “personaje()” funcionando, y por tanto, dos imágenes iguales en pantalla. Y así. Como ves, ¡es una forma

fácil de crear muchos enemigos a atacar!

Pero, ¿qué es lo que ocurre exactamente cuando se crea un proceso? Pues que se comienza a ejecutar el código que hay entre el BEGIN y el END de ese proceso. De manera, y ESTO ES IMPORTANTE, que podríamos tener ejecutándose a la vez muchos procesos, porque mientras el proceso principal crea un proceso "hijo" y las líneas de código de éste se empiezan a ejecutar, el programa principal puede tener todavía líneas por ejecutar –no en este ejemplo, pero sí es lo normal-, por lo que el ordenador empezará a ejecutar en paralelo los códigos de todos los procesos existentes en ese momento.

Miremos el código del interior del proceso "personaje()". Lo primero que hacemos es establecer valores a las variables locales predefinidas ***X***, ***Y***, ***GRAPH*** y ***FILE***. Es decir, decimos de qué FPG (***FILE***) vamos a sacar la imagen (***GRAPH***) que va a visualizar el "personaje()" en pantalla, y además decimos en qué coordenada inicial se va a situar el centro de esa imagen. Como hemos puesto una resolución de 640x480, fíjate que el gráfico aparecerá en el centro de la pantalla, inicialmente.

A continuación nos encontramos un LOOP, que comienza un bucle infinito,mientras el proceso exista, claro. Y los Ifs lo que comprueban es si alguna de las teclas de las flechas del teclado (izquierda, derecha, arriba, abajo) se ha pulsado. Si alguna condición de éstas se cumple, corrige las coordenadas del gráfico, es decir, lo mueve, porque varía los valores de las coordenadas ***X*** o ***Y***, y por tanto,reubica el centro de la imagen dada por GRAPH.

Fíjate también que si se pulsa la tecla ESC, se sale del bucle, y como no hay más código después, el proceso "personaje()" finaliza su ejecución. Pero, ¿por qué cuando "personaje()" acaba su ejecución, el programa entero también finaliza? Pues porque en ese momento no existe ningún otro proceso que se esté ejecutando, y cuando ocurre eso, el programa ya no tiene nada que hacer y acaba. Observa que el proceso el proceso "Main()" finaliza justo después de invocar a "personaje", y "personaje" finaliza al salir del bucle, con lo que tenemos que primero se ejecuta "Main()" para justo antes de "morir" pasarle el testigo al proceso "personaje()", el cual será el único proceso activo en ese momento, con lo que cuando éste acaba, como hemos visto, ya no existe ningún proceso más ejecutándose y por tanto el programa finaliza. Si existiera tan sólo una instancia de algún proceso ejecutándose, el programa no finalizaría. De todas maneras, el comportamiento de este código (que un proceso muera para dar paso a otro) no suele ser el habitual: lo normal es que varios procesos se ejecuten a la vez en paralelo porque tengan cada uno de ellos un bucle, por ejemplo.

Lo más importante a aprender y tener claro del código anterior, no obstante todo lo comentado anteriormente, es sin duda la funcionalidad de la sentencia FRAME del proceso "personaje()". Ésta sentencia es fundamental y conviene hacer unas cuantas apreciaciones. Sabemos que FRAME indica cuándo está el proceso listo para volcar la siguiente imagen a la pantalla. Lo novedoso del asunto es que, cuando nuestro programa consta de más de un proceso, hay que saber que el programa se pausará hasta que todos los procesos activos hayan llegado hasta una sentencia FRAME, momento en el cual el programa –es decir, el conjunto de procesos- reanudará su ejecución. Es decir, como podemos tener varios procesos, cada uno de los cuales con una sentencia FRAME particular, Bennu coordina el funcionamiento del programa de tal manera que espera a que todos los procesos hayan llegado, en su código particular, hasta la sentencia FRAME. El primero que llegue se tendrá que esperar al segundo y así. Esto es para que la impresión en la pantalla se haga de una manera sincronizada, con todos los procesos a la vez. Es como mandar la orden de "¡Visualizar fotograma!" para todos los procesos al unísono, de manera coordinada y al mismo tiempo. Si no se hiciera así, sería un caos, porque un proceso que tuviera poco código llegaría pronto al FRAME, mientras que uno con mucho código llegaría después y entonces se estarían visualizando fotogramas en tiempos diferentes para cada proceso, lo que sería un verdadero lío. Por lo tanto, los procesos que terminen antes su ejecución tendrán que esperarse siempre a los procesos más lentos para ir todos al mismo compás que marca el FRAME. De todo esto se puede deducir, por ejemplo, que si escribimos un bucle infinito sin sentencia FRAME el programa posiblemente se cuelgue. ¿Por qué? Porque el proceso que tenga ese bucle infinito nunca llegará al FRAME, y los demás procesos estarán pausados esperándole sin remedio. O sea que cuidado.

Para acabar de rematar el concepto de proceso, por si todavía quedan algunas dudas, vamos a modificar el ejemplo anterior para introducir otro proceso, además del que ya tenemos, de manera que aparezcan por pantalla dos objetos independientes, los cuales podrán ser movidos por teclas diferentes. El gráfico del nuevo proceso puede ser cualquiera. Puedes reutilizar el mismo gráfico que hemos utilizado para el primer proceso (no hay ningún problema para que procesos diferentes tengan el mismo valor de ***GRAPH***), o bien, que es lo más normal, utilizar otro gráfico para este nuevo proceso, en cuyo caso tendríamos que incluirlo lógicamente dentro del FPG correspondiente y usarlo como ya sabemos. Vamos a hacer esto último: crea una nueva imagen, de 30x30 por ejemplo, e inclúyela dentro del archivo "graficos.fpg", con el código 2. Lo que deberíamos de escribir para que aparecieran por pantalla estos dos procesos, y que pudiéramos controlarlos de forma independiente con teclas diferentes sería algo parecido a esto:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";

GLOBAL
    INT id1;
END
PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
    personaje2();
END

PROCESS personaje()
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        IF (key(_down)) y=y+10; END
        IF (key(_left)) x=x-10; END
        IF (key(_right)) x=x+10; END
        IF (key(_esc)) break; END
        FRAME;
    END
END

PROCESS personaje2()
BEGIN
    x=100; y=100; file=id1; graph=2;
    LOOP
        IF (key(_w)) y=y-10; END
        IF (key(_s)) y=y+10; END
        IF (key(_a)) x=x-10; END
        IF (key(_d)) x=x+10; END
        IF (key(_space)) break; END
        FRAME;
    END
END
```

Fíjate en los cambios, marcados en negrita. Lo primero que hacemos es llamar al proceso "personaje2()" desde el programa principal, para comenzar la ejecución del código que hay en su interior (en otras palabras: para que "personaje2()" se pueda ejecutar, y por tanto, entre otras cosas, visualizarse). Y luego, lo único que hemos hecho ha sido copiar prácticamente el mismo código de "personaje()" en "personaje2()". Un detalle que se ha cambiado ha sido la posición inicial. Efectivamente, comprueba que "personaje2()" ha de salir en la pantalla más a la izquierda y más arriba, porque sus ***X*** e ***Y*** son menores. Lo hemos hecho así porque si ambos procesos tuvieran la misma posición inicial, está claro que el gráfico de un proceso solaparía al otro -el último proceso que aparece en el código principal (en este caso "personaje2()") sería el que estaría por encima de los demás procesos anteriores-.

Otro detalle que se ha cambiado ha sido el valor de ***GRAPH***, para que "personaje2()" tenga asociado otra imagen, incluida no obstante en el mismo FPG, ya que si te fija, la variable ***FILE*** no se ha cambiado y continúa haciendo referencia al mismo archivo "graficos.fpg".

Y finalmente, se han cambiado las teclas de control del movimiento de "personaje2()". Si se pulsa la tecla "W", "personaje2()" se moverá hacia arriba, si se pulsa la tecla "S", se moverá hacia abajo, etc. No habría ningún problema si se asocian las mismas teclas de movimiento que a "personaje()" -los cursores-, pero entonces ¿qué pasaría?. Es fácil. Que cuando apretáramos la tecla "Left", por ejemplo, AMBOS procesos se moverían a la vez hacia la izquierda, y así con los demás cursores igual. Pruébalo.

También se ha hecho que “personaje2()” muera cuando se pulse la tecla “space”. Si ejecutas el código verás que ocurre una cosa curiosa: si pulsamos por ejemplo ESC, el proceso “personaje()” deja de visualizarse en pantalla, debido a que hemos finalizado la ejecución de su código particular porque hemos salido de su bucle LOOP particular: hemos matado el proceso y éste ha dejado de existir. Pero el programa continúa ejecutándose (a diferencia del ejemplo anterior) debido a que todavía queda un proceso en marcha: “personaje2()”. Sólo cuando matemos

“personaje2()”, pulsando SPACE (y así saliendo también de su bucle LOOP particular), no habrá ningún proceso más ejecutándose y el programa finalizará. Evidentemente, si hubiéramos pulsado primero SPACE para matar a “personaje2()”, habría desaparecido de la pantalla “personaje2()” pero “personaje()” continuaría presente, hasta que se hubiera pulsado ESC, acabando así también con el programa completo.

Importante recalcar otra vez la presencia de la orden FRAME en el nuevo proceso "personaje2()", al igual que estaba en "personaje()". Recuerda que TODOS los procesos han de tener como mínimo una orden FRAME para poderse ejecutar -y visualizar si es el caso- en cada fotograma evitando así los cuelgues del programa.

A partir de aquí, sólo nuestra imaginación nos pondrá límites a loque queramos hacer. Por ejemplo, ¿qué se tendría que modificar del ejemplo anterior para que pulsando la tecla "n", el gráfico del "personaje2()" pasara a ser el mismo que el del "personaje()", y pulsando la tecla "m" el gráfico del "personaje2()" volviera a ser el inicial? Efectivamente, tendríamos que incluir las siguientes dos líneas en el interior del bucle LOOP/END del "personaje2()" -antes de la orden Frame;-

```
if(key(_n)) graph=1; end
if(key(_m)) graph=2; end
```

Pruébalo y juega con las posibilidades que se te ocurran.

## *Alcance de las variables (datos globales, locales y privados)

Bueno, por fin ha llegado el momento de definir eso que hemos estado evitando durante todo este manual: ¿qué diferencias hay entre una variable global, local ,privada (y pública)? No le he explicado hasta ahora porque antes era necesario aprender el concepto de proceso poder entender dichas diferencias.

La diferencia entre estos tres tipos de variables está en su alcance.

***Variables GLOBALES:**

Las variables globales tienen alcance en todo el programa; esto quiere decir que cuando se declara una variable global, ésta será una variable –una posición de memoria- que puede accederse desde cualquier punto del programa. Se entiende por cualquier punto del programa el código presente en el interior de cualquier proceso definido, ya sea el proceso principal Main() como cualquier otro. Por lo tanto, cuando se dice que un dato es global es porque puede utilizarse el mismo dato en el proceso principal y en cualquiera de los otros procesos. Y por tanto, y esto es importante, cualquier cambio que se realice en el valor de esta variable desde cualquier punto del programa repercutirá inmediatemente en el resto del código. Es como si fuera un cajón compartido, el cual todos los procesos pueden abrir y cambiar su contenido cuando les tercie, de manera que cada proceso que abra el cajón se encontrará con lo que haya metido en él algún proceso anterior.Así que hay tener cuidado cuando se cambie el valor de una variable global en un proceso concreto, porque ese nuevo valor será el que vean todos los demás procesos a partir de entonces.

Tal como hemos visto, las variables globales se declaran siempre justo después de las líneas “import”, fuera de cualquier código que pertenezca a cualquier proceso (incluido el “Main()”) –cosa lógica por otra parte-. Para declarar variables globales, si son necesarias, simplemente hemos visto que es necesario escribir lo siguiente:

```
GLOBAL
        Declaraciones_y_posibles_inicializaciones;
END
```

En general, se declaran como datos globales todos aquellos que establecen condiciones generales del juego que afecten a varios procesos; un ejemplo pueden ser los puntos obtenidos por el jugador, que podrían almacenarse en la variable global “puntuación”, de modo que cualquier proceso del juego pudiera incrementarla cuando fuera necesario. También se utilizan para controlar dispositivos como la pantalla, el ratón, el joystick, el teclado o la tarjeta de sonido.

***Variables LOCALES:**

Las variables locales son variables que existen en cada proceso con un valor diferente pero que tienen el mismo nombre en todos. Por ejemplo, si se crea una variable local *puntuación*, ésta existirá y estará disponible en todos los procesos, pero si en el "proceso_1()" se le asigna un valor determinado, ese valor sólo estará asignado para el "proceso_1()". El "proceso_2()" también podrá tener propio valor para la variable *puntuación*, diferente del de la *puntuación* de "proceso_1()". Y si se cambia el valor de *puntuación* del proceso_2, no afectará para nada al valor de *puntuación* del "proceso_1()". Las variables ***GRAPH*** o ***X*** son ejemplos de variables locales (predefinidas): todos los procesos tienen una variable llamada ***GRAPH*** o ***X***, pero cada uno de ellos les asigna un valor distinto. Es decir, que cuando en un programa se utilice el nombre **X**, dependiendo de en qué proceso se utilice, se accederá a un valor numérico u otro. Cada proceso accederá con la variable local ***X*** a su propia coordenada ***X***, pues. En resumen, todos los procesos poseen el mismo conjunto de variables locales (predefinidas y definidas por nosotros), pero cada proceso asigna un valor diferente a estas variables locales.

Además, otro aspecto importante de las variables locales es que sus valores para un proceso concreto pueden ser consultados/modificados desde otros procesos diferentes. Es decir, que si desde el código de un proceso A se quiere consultar/modificar el valor de la variable GRAPH del proceso B, es posible (y viceversa)

Al igual que ocurría con las variables globales, las variables locales se declararán siempre obligatoriamente justo después de las líneas "import", fuera de cualquier código que pertenezca a cualquier proceso (incluido el "Main()"), ya que las variables locales tienen que existir para todos los procesos por igual y la única manera de conseguirlo es declarándolas fuera de cualquier proceso, de forma totalmente independiente a ellos. Si hubiera en un mismo código variables globales y locales, el orden a la hora de declararlas no es importante: da lo mismo primero declarar las globales y después las locales o viceversa, pero por convenio y tradición, primero se suelen declarar las globales y justo a continuación las locales. Para declarar variables locales, si son necesarias, simplemente habrá que escribir lo siguiente:

```
LOCAL
        Declaraciones_y_posibles_inicializaciones;
END
```

En general, se declaran como datos locales todos aquellos que se consideren informaciones importantes de los procesos. Un ejemplo puede ser la energía que le queda a un proceso (puede ser la energía de una nave, la de un disparo, la del protegonista, etc). Esta información podría almacenarse en la variable local *energía* de modo que cualquier proceso pudiera acceder y modificar su propia energía, e incluso la de los demás (por ejemplo, cuando colisionara con ellos les podría quitar energía), ya que también es posible leer y escribir los valores de una varible local de un proceso desde otro (ya veremos cómo más adelante, cuando estudiemos los identificadores de procesos) Así pues, en general este tipo de variables se usan cuando hay cosas en común en varios procesos distintos, como pueden ser energías, vidas, puntos de fuerza, de magia, o contadores. También existen muchas variables locales ya predefinidas por Bennu que sirven para aspectos muy concretos, como por ejemplo saber cúal es el gráfico de un proceso (variable local predefinida ***GRAPH***), en qué posición aparece (variables locales predefinidas ***X*** e ***Y***), cúal es su tamaño, su ángulo, su plano de profundidad, etc.

***Variables PRIVADAS:**

Por último, los datos privados son datos que se utilizan exclusivamente en un proceso del programa. Por ejemplo, si un objeto del juego requiere una variable para contar de 0 a 9 guardará este valor en una variable privada, pues los demás procesos no necesitan utilizar este valor. Es decir, que si se define en un proceso un dato privado llamado *micontador*, éste será un nombre que no significará nada dentro de otro proceso.

Es por esta razón, por ejemplo, que en el primer código de ejemplo de este capítulo se declaró como global la variable *id1*,ya que si no el ejemplo no funcionaba. Esto es porque si se declaraba privada de "Main()" (que es lo que hemos venido haciendo hasta ahora), la variable *id1* solamente tendría sentido dentro del proceso principal del programa, pero no dentro de los demás procesos, donde *id1* no significaría nada. Como en el proceso "personaje()" se utiliza la variable *id1*, ya que su valor se le intenta asignar a FILE, es necesario que *id1* sea reconocida por ambos procesos. Podría ser pues o una variable local o una global, pero está claro que ha de ser global porque *id1* solamente va a contener un valor único para todos los procesos que existan: el identificador –único- del fichero FPG cargado. No tendría ningún sentido que la variable fuera local.

Otra diferencia importante de las variables privadas respecto las locales es que en el caso de las privadas NO es posible consultar/modificar sus valores para un proceso concreto desde otros procesos diferentes. Es decir, que si desde el código de un proceso A se quiere consultar/modificar el valor de una variable privada del proceso B, NO es posible (y viceversa)

Para declarar variables privadas, si son necesarias, simplemente hemos visto que es necesario escribir antes del bloque BEGIN/END de cualquier proceso lo siguiente:

```
PRIVATE
        Declaraciones_y_posibles_inicializaciones;
END
```

Esta sección puede aparecer tanto en el proceso principal Main() como en cualquier otro proceso del programa.

En general se declaran como datos privados todos aquellos que vayan a ser utilizados como contadores en un bucle, las variables para contener ángulos o códigos identificadores secundarios, etc. Si un dato declarado como privado necesita consultarse o modificarse desde otro proceso (identificador.dato), entonces se deberá declarar dicho dato como local, (dentro de la sección LOCAL del programa) o público (ya lo estudiaremos más adelante); de esta forma, todos los procesos poseerán el dato, pudiendo acceder cada uno a su valor o al valor que tenga dicho dato en otro proceso.

En resumen:

*Los datos globales se pueden consultar/modificar desde cualquier punto del programa

*Los locales son aquellos que tienen todos los procesos (cada proceso tienen su propio valor en ellos) y se pueden consultar/modificar entre procesos.

*Los privados son los que pertenecen a un solo proceso determinado y no puede ser consultado/modificado por ningún otro proceso diferente.

.

## *Ámbito de las variables

Ya conocemos tres de los cuatro tipos de variables que existen en Bennu (globales, locales y privadas: las públicas las trataremos más adelante). Y ya se ha indicado en el apartado anterior, y por tanto, deberíamos saberlo, si es posible o no declarar cualquiera de los tres tipos de variables dentro de cualquier proceso. Pero es un ejercicio muy pedagógico e interesante comprobar qué ocurre cuando intentamos declarar diferentes variables de diferente tipo en diferentes sitios, sin tener en cuenta el lugar "correcto" para ello. Vamos a partir del ejemplo que estuvimos viendo unos párrafos más atras: lo vuelvo a escribir aquí:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";

GLOBAL
    INT id1;
END
PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
    personaje2();
END

PROCESS personaje()
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        IF (key(_down)) y=y+10; END
        IF (key(_left)) x=x-10; END
        IF (key(_right)) x=x+10; END
```

```
        IF (key(_esc)) break; END
        FRAME;
    END
END

PROCESS personaje2()
BEGIN
    x=100; y=100; file=id1; graph=2;
    LOOP
        IF (key(_w)) y=y-10; END
        IF (key(_s)) y=y+10; END
        IF (key(_a)) x=x-10; END
        IF (key(_d)) x=x+10; END
        IF (key(_space)) break; END
        FRAME;
    END
END
```

La única sección de declaraciones que vemos es la sección de variables globales, al principio de todo. ¿Tendría sentido escribir alguna sección más de variables globales dentro de algún proceso? Es decir, escribir por ejemplo:

```
PROCESS personaje()
GLOBAL
    INT varGlobalPersonaje;
END
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        ...
    END
END
```

Pues no, no tiene ningún sentido, como podrás comprobar si intentas compilar el código anterior: obtendrás un error. Y no tiene sentido porque ya hemos visto que las variables globales son como un único cajón compartido,donde todos los procesos pueden acceder a él y leer y escribir valores. Así que las variables globales no son propias de ningún proceso, y por lo tanto, insisto, la única sección de declaraciones de variables globales que habrá en nuestro programa estará antes del código de todo proceso.

Vayamos ahora a por las variables locales. ¿Dónde se declaran? ¿Fuera de los procesos o dentro de los procesos que queramos? Ya lo deberías de saber porque lo acabamos de explicar en el apartado anterior, pero piensa en lo que significa ser una variable local: es una variable de la cual TODOS los procesos tienen una copia (eso sí, con valores posiblemente diferentes). Es decir, si TODOS los procesos tienen una misma variable local "X", podemos concluir que la única sección de declaraciones de variables locales que habrá en nuestro programa estará antes del código de todo proceso, ya que precisamente es el único sitio al cual tienen acceso todos los procesos sin problemas.

¿Y las variables privadas? Tal como su nombre indica, estas variables son particulares de cada proceso :ningún proceso tiene por qué saber qué variables privadas tiene declaradas otro, y viceversa. Así que queda claro que en el caso de las variables privadas, existirá una sección de declaraciones de variables privadas dentro del código interno de cada proceso (incluido el "Main()") donde sea necesario.

Así pues, para escribir correctamente un programa que utilizara estos tres tipos de variables, tendríamos que hacer lo siguiente (tomando el código de los ejemplos anteriores):

```
Import "mod_video";
Import "mod_map";
Import "mod_key";

GLOBAL
    INT id1;
END

/*La variable local no se usa en este ejemplo, pero para declararla lo tengo que hacer
```

```
aquí,preferiblemente después de la sección de variables globales, si ésta existe.*/
LOCAL
    INT varLocal;
END

PROCESS Main()
PRIVATE
    INT varPrivadaCodigoMain; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
    personaje2();
END

PROCESS personaje()
PRIVATE
    INT varPrivadaPersonaje; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        ...
    END
END

PROCESS personaje2()
PRIVATE
    INT varPrivadaPersonaje2; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
    x=100; y=100; file=id1; graph=2;
    LOOP
        IF (key(_w)) y=y-10; END
        ...
    END
END
```

El orden en el que aparezcan las secciones globales y locales de declaraciones es irrelevante, pero por convenio se suele escribir primero la sección GLOBAL y después la sección LOCAL, si hubiere las dos.

Bien, ahora que ya sabemos dónde ubicar las distintas declaraciones de las distintas variables, podremos comprender el funcionamiento del siguiente ejemplo, que muestra algunas de las características definitorias de los tres tipos de variables comentados hasta ahora (globales, locales, privadas):

```
Import "mod_key";
Import "mod_text";

GLOBAL
          int varGlobal;
END
LOCAL
          int varLocal;
END

PROCESS Main()
PRIVATE
          int varPrivCodPrinc;
END
BEGIN
    personaje();
    personaje2();
    write_var(0,100,80,4,varGlobal);
    write_var(0,150,80,4,varLocal);
    write_var(0,200,80,4,varPrivCodPrinc);
```

```
    LOOP
                    if(key(_a)) varGlobal=varGlobal+1; end
                    if(key(_d)) varLocal=varLocal+1; end
                    if(key(_g)) varPrivCodPrinc=varPrivCodPrinc+1; end
                    FRAME;
    END
END

PROCESS personaje()
PRIVATE
          int varPrivPers1;
END
BEGIN
    write_var(0,100,100,4,varGlobal);
    write_var(0,150,100,4,varLocal);
    write_var(0,200,100,4,varPrivPers1);
    LOOP
                    if(key(_b)) varGlobal=varGlobal+1; end
                    if(key(_e)) varLocal=varLocal+1; end
                    if(key(_h)) varPrivPers1=varPrivPers1+1; end
                FRAME;
    END
END

PROCESS personaje2()
PRIVATE
          int varPrivPers2;
END
BEGIN
    write_var(0,100,120,4,varGlobal);
    write_var(0,150,120,4,varLocal);
    write_var(0,200,120,4,varPrivPers2);
    LOOP
                    if(key(_c)) varGlobal=varGlobal+1; end
                    if(key(_f)) varLocal=varLocal+1; end
                    if(key(_i)) varPrivPers2=varPrivPers2+1; end
                FRAME;
    END
END
```

Ejecuta el programa. Verás nueve 0 dispuestos en tres columnas de tres filas.¿Qué pasa si pulsamos la tecla “a”? Que aumentarán los valores de los números de la columna de más a la izquierda ¿Por qué? Si te fijas en el código, cuando pulsamos “a” lo que hacemos es aumentar el valor de *varGlobal*, y su nuevo valor lo imprimimos en tres sitios diferentes -fíjate en el **write_var()** correspondiente dentro de cada uno de los tres procesos que hay en el código-. ¿Por qué ponemos en pantalla ese valor en tres sitios? Para demostrarte que el valor de una variable global lo podemos modificar desde cualquier proceso y ese valor será el que usarán de forma compartida el resto: apreta la tecla “b” o la tecla “c” y verás que ocurre lo mismo que pulsando “a”, ya que, aunque lo hagamos desde un proceso u otro diferente, estamos modificando la misma variable *varGlobal*.

Si pulsas “d”, estarás aumentando sólo el número de la fila superior de la columna central, que se corresponde con el valor de la variable local del proceso “Main()”. Otra variable local diferente pero con el mismo nombre pertenecerá al proceso “personaje()” y otra también diferente pero homónima pertenecerá a “personaje2()”: el valor de la primera se modifica con la tecla “e” y el de la segunda con la tecla “f”, observando el correspondiente aumento en las cifras central e inferior-central, respectivamente.

Si pulsas “g” estarás modificando el valor de la variable privada del proceso principal, si pulsamos “h” el de “personaje()” y si pulsamos “i” el de “personaje2()”. En este ejemplo no se puede ver la diferencia que existe entre variables locales y privadas porque no estamos accediendo desde procesos diferentes a sus valores, (esto ya veremos más adelante cómo se hace con las variables locales), momento donde se nota la diferencia entre éstas, aunque ahora lo que sí que puedes probar es escribir por ejemplo dentro del proceso “personaje()” (después del begin) una línea como ésta:

```
write_var(0,100,200,4,varPrivPers2);
```

Verás que al intentar ejecutar el programa ocurre un error, ya que el proceso “personaje()” está intentando

acceder a una variable privada de otro proceso, cosa que no puede hacer.

Siguiendo con el ejemplo anterior, ¿qué ocurriría si tenemos dos variables distintas con el mismo nombre? ¿Ocurriría algún error o funcionaría bien el programa? De entrada, decir que es una práctica MUY poco recomendable utilizar el mismo nombre para distintas variables, porque se presta a confusión y a medio plazo es una fuente inagotable de errores y dolores de cabeza. Pero si aun así lo quieres probar, has de saber que solamente se permiten dos casos en los que dos variables diferentes pueden llamarse igual (en el resto de posibilidades se produce un error de compilación a causa de la ambigüedad en los nombres):

| |
|---|
| 1.-Dos variables privadas de distintos procesos se pueden llamar igual ya que son completamente independientes entre sí. |
| 2.-Una variable global y una variable privada se pueden llamar igual. No obstante, siempre que se haga referencia al nombre de ambas variables dentro del código del proceso donde se ha definido la variable privada homónima,dicho nombre hará referencia siempre a la variable privada, y no la global. |

A lo mejor el segundo punto es el más enredado de todos: nada más fácil que verlo en un ejemplo:

```
Import "mod_key";
Import "mod_text";

GLOBAL
          int hola=4;
END
PROCESS Main()
BEGIN
    personaje();
    personaje2();
    write_var(0,100,80,4,hola); //Se muestra el contenido de la variable global
    LOOP
                    FRAME;
    END
END

PROCESS personaje()
PRIVATE
          int hola=2;
END
BEGIN
    write_var(0,100,100,4,hola); //Se muestra el contenido de la variable privada
    LOOP
        FRAME;
    END
END

PROCESS personaje2()
BEGIN
    write_var(0,100,120,4,hola); //Se muestra el contenido de la variable global
    LOOP
        FRAME;
    END
END
```

Por cierto, no lo he comentado hasta ahora, pero una regla básica (entre otras más esotéricas) a la hora de nombrar a las variables de nuestro programa, -sean del tipo que sean-, es que su nombre no puede empezar por un dígito numérico. Es decir, variables llamadas “1var” o “9crack” son inválidas y provocarán un error de compilación. Existen más normas que prohíben la inclusión de determinados caracteres especiales en el nombre de las variables (como %, $,etc), pero a menos que seas un poco extravagante, no te deberás preocupar por esto, ya que siempre usaremos para nombrar variables caracteres alfabéticos, o en todo caso alfanuméricos.

## *Constantes

Las constantes son nombres (como los que se dan a las variables) que se utilizan como sinónimos de valores. Se pueden entender como variables que siempre tienen un mismo valor definido, un valor que no se puede cambiar. Estos valores pueden ser numeros enteros, decimales o cadena de caracteres, pero en ningún caso, a diferencia de las variables, se especificará explícitamente su tipo de datos: simplemente se asignará al nombre de la constante su valor correspondiente, del tipo que sea,y ya está.

Utilidad: se podría utilizar, por ejemplo, una constante denominada "altura_maxima" como un sinónimo permanente del valor numérico 100. Es decir, que sería indiferente utilizar "altura_maxima" o 100 en el programa. De esta manera, el código del programa sería más claro, porque se estaría informando que leyeran el programa de que el número 100 realmente equivaldría siempre a la altura máxima (de cualquier cosa u objeto del juego que fuera).

Otro uso de las constantes es el siguiente. Si en un juego se establece en varias ocasiones 3 como el número de vidas del protagonista, cuando se quiera modificar para aumentarlo o disminuirlo se tendría que buscar y sustituir ese número por todo el programa, corriendo además el riesgo de sustituir otro número 3 que pudiera aparecer en el programa para otra cosa. Una alternativa es declarar una constante denominada, por ejemplo, "maximo_vidas" como un sinónimo del valor numérico 3 y utilizar en el programa dicha constante en lugar del número; ahora, cuando se quiera cambiar este valor por otro, simplemente habrá que hacerlo una sola vez en la declaración de constante "maximo_vidas".Y ya está.

Las constantes se declaran siempre justo después de las líneas "import", fuera de cualquier código que pertenezca a cualquier proceso (incluido el "Main()"). Si hubiera alguna sección GLOBAL/END ó LOCAL/END, éstas se escribirían después de la declaración de las constantes. Para declarar constantes, si son necesarias, habrá que escribir lo siguiente:

```
CONST
        Nombre_Constante= Valor;
END
```

donde el valor puede ser un número entero (por ejemplo: 475), un número decimal (por ejemplo: 58.2) ó una cadena (por ejemplo: "hola"), pero fíjate que no se especifica en ningún sitio el tipo de la constante.

Igual que pasa con las variables, también hay constantes predefinidas. Constantes que ya conocemos son las que se pueden usar como valores del cuatro parámetro de las funciones **write()** y **write_var()**, las constantes del parámetro de flags de las funciones **map_xput()** y familia, o las constantes para los valores máximos y mínimos permitidos en los distintos tipos de variables.

## *Parámetros de un proceso

Ahora veremos ejemplos de programas con procesos que admiten parámetros, para ver cómo funcionan. En general, los procesos pueden recibir parámetros de los siguientes tipos de datos:

| | |
|---|---|
| Una variable local predefinida<br>(p.ej: X ,GRAPH,FILE,etc) | *No hay que declararlas en ningún sitio.<br>*No es necesario especificar su tipo en la cabecera del proceso |
| Una variable privada del propio proceso | *La declaración se realiza directamente en la cabecera del proceso<br>*Como consecuencia, no se ha de escribir ninguna sección PRIVATE/END en el cuerpo del proceso. |
| Una variable global | *La declaración se realiza en la sección GLOBAL/END<br>*No es necesario especificar su tipo en la cabecera del proceso |
| Una variable local | *La declaración se realiza en la sección LOCAL/END<br>*No es necesario especificar su tipo en la cabecera del proceso |
| Una variable pública | Ya veremos este caso más adelante |

Como ejemplo, ahí va un programa con un proceso que recibe cuatro parámetros diferentes de los tipos indicados en la tabla anterior (menos el tipo público) y otro proceso más que sólo recibe un parámetro de tipo privado pero que en su interior utiliza también los valores de las variables local y global declaradas al principio de todo.

```
Import "mod_key";
Import "mod_text";

GLOBAL
        float puntos;
END
LOCAL
        float energia;
END
PROCESS Main()
BEGIN
        mi_proceso(1,2.0,3.0,4);
        mi_otroproc(5);
END

PROCESS mi_proceso( x, puntos, energia, int n)
BEGIN
        write(0,0,0,0,x);
        write_var(0,0,10,0,puntos);
        write(0,0,20,0,energia);
        write(0,0,30,0,n);
        repeat
                frame;
        until(key(_esc))
END

PROCESS mi_otroproc(int n)
BEGIN
        write_var(0,0,50,0,puntos);
        write(0,0,60,0,energia);
        write(0,0,70,0,n);
        repeat
                frame;
        until(key(_esc))
END
```

Vemos que el funcionamiento es el siguiente: a la hora de invocar el proceso se ponen los valores que queramos que tengan los parámetros, y en la cabecera del proceso –en la línea del *PROCESS etc*- se escriben los nombres de las variables a las cuales se les va a asignar esos valores, en el mismo orden. Es decir, que en este ejemplo, cuando se cree "mi_proceso()", la variable local predefinida ***X*** de dicho proceso valdrá 1, la variable local *energía* valdrá 2, la variable global *puntos* valdrá 3 y una variable privada del proceso llamada *n* valdrá 4. Y con estos valores se ejecutará el código interno de ese proceso. Lo mismo ocurre con "mi_otroproc()", el cual al invocarse asignará el valor 5 a su variable privada *n* (que no tiene nada que ver con la variable privada *n* de "mi_proceso()")

Fíjate que en la cabecera de "mi_proceso()" no hemos especificado el tipo de ***X*** porque no hace falta ya que es una variable predefinida, ni tampoco lo hemos hecho con la variable *energia* ya que ya se declaró en su correspondiente sección LOCAL/END, ni tampoco lo hemos hecho con la variable *puntos* ya que ya se declaró en su correspondiente sección GLOBAL/END. Sí que hemos especificado el tipo de *n* , porque al ser privada no se ha declarado en ninguna sección anterior, por lo que aprovechamos para hacerlo ahora en la cabecera; la consecuencia es que entonces ya no es necesaria ninguna sección posterior de declaración PRIVATE/END dentro de "mi_proceso()", porque ya se ha declarado en la cabecera. El código de "mi_proceso()" lo único que hace es mostrar por pantalla los valores que se han pasado por parámetro: deberá de aparecer 1 , 2.0, 3.0 y 4.

Fíjate también, por otro lado, en la cabecera de "otro_proc()" sólo incorporamos un parámetro, el cual será una variable privada, llamada igual que el parámetro privado de "mi_proceso()", pero como son privados no tienen ninguna relación uno con otro. Y lo mostramos también por pantalla para que se demuestre que el valor de las dos *n* no

tiene nada que ver: la *n* de "mi_proceso()" hemos visto que vale 4 y la de "otro_proc()" vale 5. También volvemos a enseñar el valor de *puntos* y de *energia*. Ambas variables, al ser respectivamente global y local, pueden ser accedidas, recordemos, desde cualquier proceso, aunque no se les haya pasado por parámetro. En el caso de *puntos*, vemos que su valor es el que tenga asignado en ese momento. Como al ejecutarse "mi_proceso()" se le asignó a *puntos* el valor 2.0 (recordemos que *puntos* era uno de sus parámetros al cual le dimos un valor), ése es el valor que tiene en el momento de ejecutarse "otro_proc()", por lo que lo que veremos es otra vez un 2.0. En cambio, en el lugar que el programa nos muestra el valor de la variable local *energia* de "otro_proc()", vemos que sale un 0, cuando en "mi_proceso()" le habíamos asignado el valor 3.0. Esto ocurre porque recordemos que las variables locales, aunque se llamen igual, son independientes para cada proceso diferente, por lo que en realidad tenemos dos variables *energia*: la de "mi_proceso()" y la de "otro_proc()"; la primera sí que vale 3.0, pero la segunda no la hemos tocado para nada (no le hemos asignado ningún valor), por lo que veremos su valor por defecto, que es 0.

Con este sistema de utilizar parámetros, podemos ir creando diferentes procesos que tengan el mismo código pero que tengan distintos valores iniciales para algunas variables: un ejemplo es crear un proceso "enemigo()", que según los valores que pongamos a la hora de crearlo tenga asociado un gráfico u otro, una posición u otra, una energía u otra, creando así múltiples enemigos que son ligeramente diferentes mediante un único código. Un ejemplo:

```
Import "mod_video";
Import "mod_map";

PROCESS Main()
BEGIN
          mi_proceso(60,100,255);
          mi_proceso(160,200,0);
          //Más instrucciones
END

PROCESS mi_proceso(x,y, byte red)
PRIVATE
          int n;
END
BEGIN
          graph=new_map(30,30,32);
          map_clear(0,graph,rgba(red,255,0,255));
          FROM n=0 TO 99;
                    x=x+2;
                    y=y+1;
                    frame;
          END
END
```

Este ejemplo lo que hace es crear dos instancias de un proceso llamado "mi_proceso()", el cual tiene tres parámetros, y ya está. Si luego nos fijamos en el código de este proceso, podemos ver que lo que representan los dos primeros parámetros es la posición inicial x e y (porque así está escrito en la cabecera del proceso) del gráfico asociado a cada instancia concreta del proceso, con lo que las dos instancias que hemos generado ya vemos que aparecerán inicialmente en posiciones diferentes. El gráfico asociado a cada instancia (variable ***GRAPH***) se obtiene a partir de generarlo en memoria con **new_map()** y pintarlo con **map_clear()** Y lo que representa el tercer parámetro es la cantidad de color rojo que se utilizará precisamente para pintar cada instancia concreta del proceso: la primera instancia tendrá el máximo de rojo (obteniéndose en total gracias a **rgba()** un color amarillo) y la segunda instancia tendrá el mínimo de rojo (obteniéndose en total gracias a **rgba()** un color verde). Fíjate que de los tres parámetros, sólo ha sido necesario especificar en la cabecera del proceso su tipo al tercero, porque los otros dos ya son variables predefinidas (locales) Finalmente, una vez definida la posición inicial y el color del gráfico, ambas instancias (porque ambas son exactamente el mismo proceso, recordemos, y ambas ejecutan exactamente el mismo código a pesar de que tengan diferentes los valores iniciales pasados por parámetro) entran en un bucle el cual él solo va moviendo la coordenada del centro de su dibujo respectivo hacia la derecha y para abajo, hasta que finalmente el contador *n* (cada instancia el suyo) llega a 99, momento en el cual ambas instancias de "mi_proceso()" mueren.

## *Código identificador de un proceso. La variable local predefinida ID y la orden "Get_id()"

Un código identificador es un número entero diferente de 0 (y mayor siempre que 65536) que Bennu

asigna automáticamente a cada instancia de proceso en el momento de su creación, y lo identifica individualmente mientras el juego se está ejecutando. Los ID son como el DNI de cada uno de los procesos individuales que en un momento determinado estén ejecutándose en nuestro juego.

Que un ID sea asignado a un proceso en tiempo de ejecución –es decir, cuando el juego se ejecuta y no antes-, significa que no podemos saber cuáles serán esos números mientras programamos ni tampoco podremos asignarlos o cambiarlos. De todas formas, es muy sencillo obtener ese valor: está almacenado en la variable local ***ID*** de cada proceso y también –importante- es el valor de retorno que devuelven los procesos cuando son llamados. Esto último quiere decir que cuando se invoca a un proceso (generándose una instancia particular), ese proceso devuelve un número entero , el cual podemos recoger en alguna variable entera cualquiera para poder hacer así referencia a esa instancia concreta a lo largo de todo el código posterior (para matarla,o dormirla, o lo que necesitemos hacer con esa instancia).

Los ID son fundamentales a la hora de interactuar con procesos, ya que gracias a su ID podemos leer y modificar las variables locales y públicas de un proceso (las privadas no) desde otro. Por lo tanto, podríamos mover un proceso desde otro, cambiar su gráfico, rotarlo o hacer cualquier cosa que podamos hacer mediante variables locales.

Pongamos un ejemplo ahora (necesitarás un FPG llamado "graficos.fpg" con al menos dos imágenes en su interior) que es importante y nos va a servir para dos cosas: 1º) Para aclarar un poco las diferencias entre estos los tres tipos de variables vistos hasta ahora (globales, locales, privadas); 2º) Para observar la utilización de los ID de los procesos para manipular los valores de las variables locales:

```
Import "mod_video";
Import "mod_map";
Import "mod_text";

global
  int varglobal=20;
  int idproc1, idproc2;
  int idfpg;
end
local
          int varlocal;
end

process Main()
begin
          set_mode(640,480,32);
          idfpg=load_fpg("graficos.fpg");
          idproc1=proceso1();
          idproc2=proceso2();
          write(0,200,20,4,"El identificador de proceso1: " + idproc1);
          write(0,200,40,4,"El identificador de proceso2: " + idproc2);
          loop
                    frame;
          end
end

process proceso1()
private
          int varprivada1=1;
end
begin
file=idfpg;
graph=1;
x=200;
y=300;
varlocal=10;
write(0,200,80,4,"El identificador de proceso1 visto desde él mismo: " + id);
write(0,200,100,4,"La variable local de proceso1 vista desde él mismo: "+ varlocal);
//write(0,200,140,4,"La variable local de proceso2 vista desde proceso1: "+ idproc2.varlocal);
write(0,200,120,4,"La variable privada de proceso1 vista desde él mismo: " +
varprivada1);
          loop
               varglobal=varglobal+1;
               frame;
          end
```

```
end

process proceso2()
begin
file=idfpg;
graph=2;
y=100;
varlocal=20;
write(0,200,180,4,"El identificador de proceso2 visto desde él mismo: " + id);
write(0,200,200,4,"La variable local de proceso2 vista desde él mismo: "+ varlocal);
write(0,200,220,4,"La variable local de proceso1 vista desde proceso2: "+ idproc1.varlocal);
//write(0,200,240,4,"La variable privada de proceso1 vista desde proceso2 : " +
idproc1.varprivada1);
          loop
                    x=varglobal;
                    frame;
          end
end
```

En este código hay mucha chicha. Por un lado tenemos una variable global llamada *varglobal*, la cual tiene inicialmente un valor de 20. Si te fijas, desde "proceso1()" se accede a ella modificando en cada iteración (por tanto, en cada fotograma) su valor: como es global podemos hacerlo. Pero además, desde "proceso2()" también se accede a ella en cada iteración, haciendo que el valor de la posición x de su gráfico coja el valor que tiene en ese momento *varglobal*. Como las variables globales son únicas para todo el programa, si "proceso1()" ha aumentado en 1 su valor, esto repercutirá en que en ese mismo fotograma la posición horizontal de "proceso2()" será de un pixel más a la derecha.

Fíjate también que mostramos por pantalla los identificadores de "proceso1()" y "proceso2()" de dos maneras distintas: por un lado, en el programa principal a partir de las variables (globales, para que se puedan usar cada una de ellas con su valor único en todo el programa) *idproc1* y *idproc2*, las cuales recogen los identificadores de las dos instancias en el momento de su creación. Por otro lado, dentro de cada proceso, hacemos uso de una variable local predefinida -al igual que lo son las variables ***X,Y,GRAPH*** o ***FILE***, ya conocidas- llamada ***ID***. Esta variable ***ID*** almacena, para cada instancia de proceso particular, su propio identificador. En nuestro ejemplo hacemos uso de ambos métodos para obtener y mostrar por pantalla los identificadores de ambos procesos, con lo que aparecerán repetidos.

Por otro lado, tenemos una variable local llamada *varlocal*. Esto quiere decir que todos los procesos del programa tienen una variable llamada así. Si te fijas, en "proceso1()" le asigno el valor de 10 y en "proceso2()" le asigno el valor de 20. El punto interesante está en que es posible acceder/modificar al valor de *varlocal* de "proceso1()" desde "proceso2()" así como al valor de *varlocal* de "proceso2()" desde "proceso1()". Y esto lo conseguimos gracias a los identificadores de cada proceso. Acabamos de decir que cuando se llama a un proceso, éste devolverá siempre el identificador de la instancia creada, que no es más que un número entero único en ese momento. En el código de ejemplo nos aprovechamos de este hecho y dicho identificador lo recogemos en el instante que creamos cada proceso, almacenándolo en una variable (en nuestro caso, *idproc1* e *idproc2*).

Pero a partir de aquí, cuando ya tenemos guardado en estas variables el identificador de cada proceso, ¿cómo accedemos al valor de alguna de sus variables locales? Pues esto se hace en general con la sintaxis:

***identificadorProceso.nombreVariableLocal***

...que es lo que hemos hecho en el ejemplo: fíjate que en "proceso2()" logro saber cuál es el valor de *varlocal* de "proceso1()" gracias a la notación "idproc1.varlocal".

Si eres perspicaz te habrás fijado en un detalle: la línea que nos tenía que imprimir en pantalla el valor encontrado desde "proceso1()" de la *varlocal* de "proceso2()" está comentada...Si pruebas a descomentarla verás que el programa al ejecutarse lanza un error desagradable (“Runtime error – Process 0 not active” o similar) y se cierra. Y esto es por una razón sutil: ocurre que hasta que todos los procesos no llegan por primera vez a la línea frame; -al primer fotograma-, cada proceso no es consciente de la existencia de los demás procesos que han sido creados posteriormente a él. Es decir, en el código principal primero creamos a "proceso1()" y luego a "proceso2()". Bien, hasta que "proceso1()" no llegue a ejecutar el primer frame (y con él, todos los demás procesos,claro), dentro de su código no se podrá hacer referencia a ningún otro proceso que haya sido creado después de él, porque no sabe de su existencia.

Prueba un experimento: intercambia el orden de creación de los procesos (o sea, pon primero la línea que

crea "proceso2()", antes que "proceso1()" y descomenta la línea problemática. Verás que continúas teniendo el mismo problema, pero ahora causado por otra línea: la que pretende acceder desde "proceso2()" al valor de *varlocal* de "proceso1()". Y esto, ¿cómo lo solucionamos?. Una manera sería poner las referencias que se hagan a los procesos posteriores al actual después de la primera línea frame;. Es decir, si cortas la línea problemática y la pegas justo después del frame del mismo "proceso1()" -y la descomentas-, verás que al ejecutar el programa funciona perfectamente. De todas maneras, la solución óptima es utilizar un bloque DECLARE/END, cosa que veremos posteriormente.

Por último, tenemos en "proceso1()" la variable privada *varprivada1*. Podríamos intentar acceder desde "proceso2()" al valor de *varprivada1* utilizando la misma notación por ejemplo que para las variables locales (y de hecho, hay una línea comentada que lo haría), pero si la descomentas verás que se produce un error sin solución: no se reconoce *varprivada1* porque las variables privadas sólo se pueden usar dentro del proceso donde se han definido.

Hasta ahora hemos visto cómo averiguar el código identificador de una instancia de un proceso concreto (bien obteniéndolo a partir del valor devuelto en la creación de ésta, o bien mediante la variable local ***ID*** para esa instancia de proceso en cuestión). Pero existen ocasiones en que nos puede interesar obtener no uno solo, sino todos los códigos identificadores de los procesos activos que son de un tipo concreto. Es decir, obtener los Ids de todos los procesos activos que sean “enemigos”, o todos los procesos activos que sean “disparo”, o “bola”, o “proceso1”,etc... Bennu aporta una función que nos facilita la vida en este aspecto: la función **get_id()**, definida en el módulo “mod_proc”.

| **GET_ID(TIPO)** |
|---|
| Esta función, definida en el módulo “mod_proc”, devuelve el identificador del primer proceso activo que encuentre que corresponda al tipo especificado. Si no encontrase ninguno, esta función devolverá 0. En las siguientes llamadas a esta función que se realicen utilizando el mismo parámetro, **get_id()** devolverá sucesivamente los identificadores de los siguientes procesos de ese tipo que se encuentren en memoria, hasta que ya no quede ninguno, en cuyo caso devolverá 0.<br><br>Es posible llamar a **get_id()** con un parámetro de tipo de proceso 0. En este caso, **get_id()** devolverá los identificadores de todos los procesos en memoria, de cualquier tipo.<br><br>**Get_id()** no devolverá nunca identificadores de procesos muertos o marcados para matar el próximo frame mediante la función **signal()** -la veremos enseguida-<br><br>El estado de **get_id()** se reinicia cada frame, con lo que si los procesos del tipo buscado han variado (han aparecido nuevos, han muerto algunos), **get_id()** lo reflejará..<br><br>PARÁMETROS:<br><br>INT TIPO : Un tipo de proceso, dado por la expresión *TYPE nombre_del_proceso_en_el_código* |

Un ejemplo de esta función puede ser el siguiente:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_rand";
Import "mod_proc";

global
  int IDP;
  int cont;
  int png1,png2;
end

process main()
begin
  set_mode(320,200,32);
  png1=new_map(30,30,32); map_clear(0,png1,rgb(255,0,0)); //Cuadrado rojo
  png2=new_map(30,30,32); map_clear(0,png2,rgb(0,255,0)); //Cuadrado verde
  from cont=1 to 10 ; //Creo y muestro 10 instancias de los procesos rojo y verde
    cuadrojo(rand(0,299),rand(0,199),png1);
    cuadverde(rand(0,299),rand(0,199),png2);
  end
```

```
  frame; //Para ver los 10 cuadrados rojos y los 10 verdes en su posición inicial
  repeat
/*Voy del 1 al 10 porque sé que hay 10 procesos de tipo cuadrado verde y por tanto, he de
ejecutar la función get_id 10 veces para encontrar sus 10 IDs*/
    from cont=1 to 10;
/*En cada iteración IDP valdrá el identificador de un proceso "cuadverde" distinto.*/
      IDP=get_id(type cuadverde);
/*Este if hace simplemente que los cuadrados verdes se detengan en el extremo derecho de
la pantalla y no vayan más allá. Hacemos uso de la notación ya aprendida para acceder a
variables locales de un proceso cualquiera*/
      if(IDP.x < 320) IDP.x=IDP.x + 10; end
    end
    frame;
  until(key(_esc))
  exit(); //No hemos visto esta orden todavía: sólo sirve para acabar el programa bien.
end

process cuadverde(x,y,graph)
begin
  loop
    frame;
  end
end

process cuadrojo(x,y,graph)
begin
  loop
    frame;
  end
end
```

Puedes comprobar, si ejecutas el código, que lo que ocurre es que inicialmente tenemos diez procesos de tipo “cuadverde” y diez procesos de tipo “cuadrojo”. Y únicamente los de tipo “cuadverde” se mueven hacia la derecha: los de tipo “cuadrojo” se mantienen inmóviles en sus posiciones iniciales. Esto es porque la función get_id() sólamente devuelve los identificadores de los procesos “cuadverde”, y por tanto, sólo variamos los valores de la variable local X de dichos procesos.

## *Árboles de procesos

Otro concepto que deberías tener claro trabajando con procesos en Bennu es el de los árboles “genealógicos” de procesos. Si un proceso a partir de su código crea otro proceso, dentro del código del segundoe siempre se podrá hacer referencia a su proceso creador utilizando una variable local llamada ***FATHER*** –padre- la cual guarda su código identificador, y, desde el código de ese proceso padre, el proceso creado podrá ser identificado con su ID guardado en otra variable local: ***SON*** –hijo-, siempre que, eso sí, éste sea el último proceso creado Es decir, si el padre genera muchos hijos, sólamente el último de ellos podrá ser llamado Son en el código del padre. Resumiendo: un proceso Father es aquel que crea a otro, el cual será el proceso Son respecto el primero hasta que Father genere un nuevo hijo, el cual entonces será el nuevo proceso Son. Si un proceso no creara otros procesos, no tendrá ningún hijo y ***SON*** valdrá 0.

También podemos utilizar, desde el código de un proceso cualquiera, la variable local ***BIGBRO*** -hermano mayor- para identificar al último proceso que el padre del proceso actual creó antes que éste, (la cual valdrá 0 si no se había creado ningún proceso anterior al actual). Y ***SMALLBRO*** servirá para referenciar al primer proceso que el creador del proceso actual creó después de éste, (y también valdrá 0 si no se ha creado ningún proceso posterior al actual).

Así pues, resumiendo: en un proceso “proceso1()” cualquiera, la variable ***FATHER*** guarda el código identificador del proceso que llamó a “proceso1()”, la variable ***SON*** guarda el código identificativo de la última instancia de “proceso1()” creada, la variable ***BIGBRO*** guarda el código identificador de la instancia que el padre de “proceso1()” creó justo antes que la instancia actual y ***SMALLBRO*** guarda el de la instancia del proceso que el padre de “proceso1()” creó justo después de la instancia actual. Tranquilo, con los ejemplos se verá más claro todo este lío familiar.

El primer proceso que se ejecuta siempre es el proceso principal, llamado “Main()”, el cual es el Father de todos los procesos, porque él es el que crea (llama) a todos los demás procesos que se van a ejecutar. Puesto que estos procesos tienen el mismo padre, se puede decir que son hermanos. A su vez, estos procesos pueden llamar a otros procesos, por lo que se convertirán a su vez en padres de estos procesos, que serán sus hijos y hermanos entre ellos,etc.

Es importante tener en cuenta que si muere el Father, todos hijos serán “adoptados” por su abuelo: es decir, el abuelo pasaría a ser el nuevo Father. Si no existe proceso abuelo, los procesos hijos de tener vinculación entre ellos (dejan de ser “hermanos”) y van cada uno por su lado, huérfanos. Ojo con este detalle porque puede llevar a confusión si no se tiene en cuenta adecuadamente.

Otra utilidad fundamental de este sistema de árboles de procesos es la posibilidad de acceder a variables locales del proceso deseado. Por ejemplo, usando el identificador ***FATHER***, podemos acceder desde el código de cualquier proceso hijo a variables como *father.x* o *father.y*, (recordemos la notación de acceso a variables locales de un proceso cualquiera) que nos devolverán en todo momento la posición en pantalla del proceso padre (muy útil por ejemplo si queremos seguirle). De esta manera, con el uso de las variables ***FATHER, SON, BIGBRO*** *y* ***SMALLBRO*** podemos tener controlada a toda la parentela cercana de un proceso con facilidad, consultando o modificando los valores que se deseen de ***X,Y,GRAPH,FILE***...y otras variables locales que iremos viendo (como ***ANGLE,SIZE,Z***,etc).

Un ejemplo que ilustra el significado de estas variables locales predefinidas de familia de procesos podría ser éste:

```
Import "mod_text";

/*
Si father crea son1, después son2 y después son3, entonces:
  son1.bigbro = 0
  son2.bigbro = son1
  son3.bigbro = son2
  son1.smallbro = son2
  son2.smallbro = son3
  son3.smallbro = 0
  father.son = son3
*/
global
   int count=0;
end
local
   int var1,var2;
end

process Main()
begin
   myProcess1();
end

process MyProcess1()
begin
   var1= 777;
   var2= 111;
   MyProcess2();
   //Imprime el valor de var1 del proceso actual (MyProcess1), ya que el nieto de su
abuelo es él mismo
   write(0,100,20,0, son.son.father.father.var1);
   loop frame;end
end

process MyProcess2()
private
    int i;
end
begin
   i=MyProcess3();
   MyProcess3();
   MyProcess3();
   MyProcess3();
```

```
   //Imprime el valor de var1 del proceso smallbro del primer proceso MyProcess3 creado
   write(0,100,40,0, i.smallbro.var1 );
   //Imprime el valor de var1 del proceso bigbro del último proceso MyProcess3 creado
   write(0,100,60,0, son.bigbro.var1 );
   //Por curiosidad:¿qué sería entonces  "son.bigbro.bigbro.var1"?
   loop frame;end
end

process MyProcess3()
begin
   count++;
   var1=count;
   //Imprime el valor de var2 del abuelo del proceso actual (es decir, MyProcess1)
   write(0,100,80,0, father.father.var2 );
   loop  frame; end
end
```

Fíjate en los valores mostrados por pantalla y deduce, a partir del código, por qué son los que son.

Acabamos de ver que la variable ***SON*** siempre guarda el identificador del último proceso generado por un proceso "padre" determinado. Pero, ¿cómo podríamos hacer para conocer TODOS los procesos hijos de ese proceso "padre", y no sólo el último de todos? Una manera sería ir "cabalgando" desde el proceso ***SON*** hasta su inmediato hermano mayor con ***BIGBRO***, y una vez allí, volver a "cabalgar" hacia el siguiente proceso, hermano mayor del hermano mayor de ***SON***, y así hasta que ya no se obtenga ningún identificador de ningún proceso (es decir, el valor 0). Esta idea se ve más clara en el siguiente ejemplo (no te preocupes por el -todavía desconocido- comando **say()**: sólamente has de saber por ahora que es similar a **write()** pero el texto en vez de escribirlo por la pantalla del programa, lo escribe en la consola del intérprete de comandos (es decir, dentro de la "ventanita negra")

```
import "mod_proc";
import "mod_say";

Process Main()
Private
          int sonID;
End
Begin
          Proc();
          Proc();
          Proc();

          sonID = son;
          while(sonID !=0)
                    say("Son: " + sonID);
                    sonID = sonID.bigbro;
          end
          exit();
End

Process Proc()
Begin
          Loop
                    frame;
          End
End
```

Otra manera de recorrer todos los hijos, diferente de la manera anterior, y bastante más "profesional", es implementando algún árbol de recorrido en preorden o similar, pero este aspecto tan avanzado de programación se escapa por completo de los objetivos de este texto.

## *Señales entre procesos: orden "Signal()"

Un proceso A puede enviar un aviso, una señal a otro proceso B. Para ello, el proceso A ha de ejecutar la

orden **signal(),** definida en el módulo "mod_proc". Este comando tiene dos parámetros: el primero será el código identificador del proceso destino (proceso B); el segundo es la señal que se mandará, la cual se ejecutará inmediatamente, sin esperar al siguiente frame (excepto cuando la señal es enviada al mismo proceso que la generó -cuando se "autoenvía" la señal-, circunstancia en la que sí se espera al siguiente frame para hacer efectiva la señal).

En el módulo "mod_proc" hay definida una serie de constantes que sirve para dar nombre a la señal concreta que será lanzada mediante **signal()**. Algunas de ellas son:

| | | |
|---|---|---|
| **S_KILL** | Equivale a 0 | Mata al proceso, por lo que su código dejará de ejecutarse y desaparecerá de la memoria y de la pantalla para siempre.Una vez muerto no se podrá acceder a sus variables locales ni modificarlo, pues ya no existe. |
| **S_WAKEUP** | Equivale a 1 | Despierta a los procesos dormidos o congelados (ver siguientes señales) de manera que sigan ejecutando su código donde lo dejaron. |
| **S_SLEEP** | Equivale a 2 | Duerme el proceso, por lo que su código no se ejecutará ni aparecerá en pantalla, pero el proceso seguirá existiendo y por tanto se podrá seguir accediendo a sus variables locales y modificarlas desde otros procesos. |
| **S_FREEZE** | Equivale a 3 | Congela el proceso, por lo que su código no se ejecutará pero el gráfico seguirá quieto en pantalla (y los demás procesos podrán detectar colisiones con su gráfico), y también se podrá seguir accediendo y modificando sus variables locales. |
| **S_KILL_TREE** | Equivale a 100 | Igual que S_KILL, pero además de actuar sobre el proceso destino que pongamos en **signal()**, mata a toda su descendencia (procesos hijos, "nietos",etc) |
| **S_WAKEUP_TREE** | Equivale a 101 | Igual que S_WAKEUP, pero además de actuar sobre el proceso destino que pongamos en **signal()**, despierta a toda su descendencia |
| **S_SLEEP_TREE** | Equivale a 102 | Igual que S_SLEEP, pero además de actuar sobre el proceso destino que pongamos en **signal()**, duerme a toda su descendencia |
| **S_FREEZE_TREE** | Equivale a 103 | Igual que S_FREEZE, pero además de actuar sobre el proceso destino que pongamos en **signal()**, congela a toda su descendencia |

La razón de la existencia de las señales *_TREE es fácil de ver. Si se mata a un proceso, toda su descendencia se queda huérfana, con lo que los diferentes procesos hijo pierden su relación directa entre ellos (los hermanos ya dejan de tener una relación común porque el padre ha muerto y cada uno va por su lado), y mandar señales a estos procesos se convierte en tarea complicada. Por eso es conveniente en más de una ocasión eliminar a todo el árbol. Imagínate la siguiente situación en un juego de matar enemigos: tenemos 20 instancias de disparos generados por un (o varios) procesos enemigo. Si te matan y el juego se acaba, todos los enemigos y disparos ya no deberían de estar en pantalla: tienes que terminar sus procesos. Podrías hacerlo uno a uno, pero lo mejor es terminar el árbol entero (en este caso, empezando por el -o los- procesos enemigos), ya que así, siempre que lo hagas de la manera adecuada, terminas todos sus procesos hijo de un plumazo. Para eso existen las señales _TREE, que afectan a todo el árbol que crea el proceso que le pasamos por parámetro en **signal()**. Este hecho se puede usar en unión con el hecho de que como primer parámetro de la orden **signal()** también se pueden poner directamente las variables ***FATHER,SON***,etc (recordemos que su valor no es más que el identificador de un proceso determinado, como cualquier otro).

Hasta ahora he comentado que la orden **signal()** era capaz de enviar una señal a una instancia de un proceso especificado. Pero eso no es todo. Esta orden también se puede utilizar para enviar la misma señal a la vez a un conjunto amplio de instancias de procesos de un tipo concreto. Es decir, si estás programando un matamarcianos, es posible que necesites en algún momento matar a todos los enemigos de golpe (porque el jugador ha conseguido un mega-cañón ultrasónico que los deja a todos fritos de golpe con su onda expansiva). Recuerda que cada enemigo es una instancia concreta (con su ID propio) de un proceso llamémosle "enemigo()". Si utilizáramos la orden **signal()** como hasta ahora se ha explicado, tendríamos que ingeniarnos un complicado sistema para recoger todos los identificadores de los diferentes enemigos existentes en el juego e ir llamando dentro de un bucle a la orden **signal()** pasándole en cada iteración un ID diferente, para ir matando a los enemigos uno a uno. ¿No sería más fácil enviar una señal a todos los procesos que sean instancias de un proceso determinado? Algo así como "enviar esta señal a todos los procesos de tipo 'enemigo'", en nuestro caso. Y ahorrarnos un montón de faena...Pues se puede, con la misma orden **signal()**. Si quieres saber cómo se haría para enviar una señal a un conjunto de procesos del mismo tipo, lo único que hay que escribir es como primer parámetro de la orden **signal()** –el destinatario de la señal-, en vez de un código identificador, la sentencia

*TYPE nombre_del_proceso_en_el_código*. Es decir, que si quisiéramos destruir todos los enemigos de golpe, tendríamos que escribir *signal(TYPE enemigo,s_kill);*

Un ejemplo del envío de señales mediante signal() podría ser el siguiente.

```
Import "mod_proc";
Import "mod_video";
Import "mod_map";
Import "mod_text";
Import "mod_key";

global
  int IDB; //Identificador del proceso "Bola()"
/*Valdrá 1 si el proceso "Bola()" está vivo y 0 si no. Nos servirá para que, en el
momento que "Bola()" se detecte que esté muerta, se vuelva a generar otro proceso
"Bola()" y se pueda continuar el programa. Un proceso dejará de estar vivo SÓLO si ha
recibido una señal de tipo KILL.*/
  int bolaActiva;
end

process main()
begin
  set_mode(640,480,32);
  write(0,10,30,3,"1) S_KILL");
  write(0,10,40,3,"2) S_FREEZE");
  write(0,10,50,3,"3) S_SLEEP");
  write(0,10,60,3,"4) S_WAKEUP");
  write(0,10,70,3,"5) S_KILL_TREE");
  write(0,10,80,3,"6) S_FREEZE_TREE");
  write(0,10,90,3,"7) S_SLEEP_TREE");
  write(0,10,100,3,"8) S_WAKEUP_TREE");
  IDB=Bola(); bolaActiva=1;
  repeat
      if(key(_1) && bolaActiva==1) signal(IDB,s_kill);bolaActiva=0;end
      if(key(_2) && bolaActiva==1) signal(IDB,s_freeze);end
      if(key(_3) && bolaActiva==1) signal(IDB,s_sleep);end
      if(key(_4) && bolaActiva==1) signal(IDB,s_wakeup);end
      if(key(_5) && bolaActiva==1) signal(IDB,s_kill_tree);bolaActiva=0;  end
      if(key(_6) && bolaActiva==1) signal(IDB,s_freeze_tree);end
      if(key(_7) && bolaActiva==1) signal(IDB,s_sleep_tree);end
      if(key(_8) && bolaActiva==1) signal(IDB,s_wakeup_tree);end
/*Este if lo único que hace es, si la bola de arriba está muerta, esperar un poco y
volver a crear una nueva bola. No te preocupes por el parámetro -no visto todavía- que
tiene la orden frame: sólo decir que sirve para pausar un poquitín el juego de manera que
se note que durante unos instantes no hay bola de arriba.*/
      if(bolaActiva==0)
            frame(2000);
            IDB=Bola(); bolaActiva=1;
      end
      frame;
  until(key(_esc))
  exit();
end

process Bola()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
  x=180;
  y=40;
  Bola2(); //Nada más generarse "Bola()", ésta generará "Bola2()"
  loop
      while(x<600) x=x+5; frame; end
      while(x>180) x=x-5; frame; end
  end
end

process Bola2()
begin
```

```
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
  x=180;
  y=80;
  loop
      while(x<600) x=x+5; frame; end
      while(x>180) x=x-5; frame; end
  end
end
```

Si ejecutas este código, lo que podrás ver será dos gráficos que no paran de moverse constantemente de un lado a otro de la pantalla, y un menú con 8 opciones para aplicar cada una de las 8 señales existentes a alguno (o todos) de los procesos activos. El gráfico de arriba es un proceso ("Bola") y el gráfico de abajo es otro ("Bola2"). El de arriba es padre del de abajo, al menos al principio de la ejecución del programa.Estudiemos lo que pasa y su relación con el código fuente de la siguiente manera:

*Si pulsamos la tecla "2", siempre que "Bola()" esté vivo, lo congelamos. Se puede comprobar que el gráfico correspondiente se para y deja de moverse, pero al gráfico de debajo no le pasa nada porque es otro proceso diferente que no resulta afectado por la señal.

*Si seguidamente pulsamos la tecla "4", siempre que "Bola()" esté vivo, lo descongelamos ó despertamos, según el caso. En este caso, al descongelarlo, lo que veremos es que el gráfico comienza otra vez a moverse de forma normal desde el sitio donde permanecía congelado.

*Si ahora pulsamos la tecla "3", siempre que "Bola()" esté vivo, lo dormimos. Se puede comprobar que el gráfico correspondiente desaparece de la pantalla, pero al gráfico de debajo no le pasa nada porque es otro proceso diferente que no resulta afectado por la señal. Si volvermos a pulsar otra vez la tecla "4", veremos que despertamos a "Bola()", con lo que volveremos a ver el gráfico en pantalla y éste comenzará otra vez a moverse de forma normal desde el sitio donde desapareció.

*Si pulsamos la tecla "6", siempre que "Bola()" esté vivo, congelamos el proceso "Bola()" y toda su descendencia, que en este caso es sólo el proceso "Bola2()". Es decir, en definitiva congelamos dos procesos con una sola orden **signal()**. Podemos comprobar como en este caso, ambos gráficos se paran y dejan de moverse.

*Si seguidamente pulsamos la tecla "8", siempre que "Bola()" esté vivo, descongelamos ó despertamos, según el caso, al proceso "Bola()" y toda su descendencia, que en este caso es sólo el proceso "Bola2()". Lo que veremos como resultado es que ambos gráficos comienzan a la vez a moverse de nuevo de forma normal desde el sitio donde permanecían congelados, como si no hubiera pasado nada.

*Si ahora pulsamos la tecla "7", siempre que "Bola()" esté vivo, dormimos el proceso "Bola()" y toda su descendencia, que en este caso sólo es "Bola2()". Podemos comprobar como en este caso, ambos gráficos desaparecen de la pantalla.Si volvermos a pulsar otra vez la tecla "8", veremos que despertamos a "Bola()" y a toda su descendencia (o sea, "Bola2()" también), con lo que volveremos a ver ambos gráficos en pantalla y éstos comenzarán otra vez a moverse de forma normal desde el sitio donde desaparecieron cada uno.

*Si pulsamos la tecla "5", siempre que "Bola()" esté vivo, matamos a éste y toda su descendencia (es decir, al proceso "Bola2()" también lo matamos). Como resultado, ambos procesos desaparecerán de la pantalla. No obstante, al cabo de unos instantes, el código fuente del ejemplo ha sido escrito de tal manera que se vuelve a generar un nuevo proceso "Bola()" -y éste, un nuevo proceso "Bola2()"- volviendo a empezar de nuevo.

*Si pulsamos la tecla "1", siempre que "Bola()" esté vivo, lo matamos. Se puede comprobar que el gráfico correspondiente desaparece, pero al gráfico de debajo no le pasa nada porque es otro proceso diferente que no resulta afectado por la señal. No obstante, al cabo de unos instantes, el código fuente del ejemplo ha sido escrito de tal manera que se vuelve a generar un nuevo proceso "Bola()" -y éste, un nuevo proceso "Bola2()"-, con lo que tendremos tres gráficos en pantalla: el perteneciente a los nuevos "Bola()" y "Bola2()", y el perteneciente al antiguo "Bola2()" que ha quedado huérfano. Si ahora pruebas por ejemplo de pulsar la tecla 5 (la señal S_KILL_TREE) verás que sólo morirán los dos procesos acabados de generar, porque son los únicos que están emparentados: el antiguo "Bola2()" tiene un padre que ya no existe, y por tanto, no pertenece a la misma familia que los nuevos. Esto implica que, tal como está escrito el código de ejemplo, no le podemos enviar ninguna señal a este proceso huérfano,por lo que permanecerá en pantalla a perpetuidad, independientemente de la creación/destrucción de nuevas familias de procesos.

Otro ejemplo,similar al anterior, de utilización de señales dirigidas a toda la descendencia completa de un proceso padre (todo un árbol), es el siguiente:

```
Import "mod_proc";
Import "mod_video";
Import "mod_map";
Import "mod_text";
```

```
Import "mod_key";

Global
   Int mapCirculo;
   Int idCirculator;
   int texto;
   Int pausado = 0;
end

Process Main()
Begin
   set_mode (640,480,32);
   mapCirculo = new_map(40,40,32); map_clear(0,mapCirculo,rgb(0,0,255));
   idCirculator = circulator();
   write(0,0,0,0,"P para pausar todos los hijos");
   write(0,0,10,0,"O para quitar la pausa de todos los hijos");
   write(0,0,20,0,"S para pausar el último hijo ('Son')");
   write(0,0,30,0,"A para quitar la pausa del último hijo");
   write(0,0,40,0,"ESC para salir");
   Loop
      If (key(_esc)) exit(); End //El comando exit() lo veremos en siguientes apartados
      If (key(_p) AND NOT pausado)
         signal(idCirculator,s_freeze_tree);
         texto = write(0,320,240,4,"Pausa total");
         pausado = 1;
      End
      If (key(_o) AND pausado)
         signal(idCirculator,s_wakeup_tree);
         delete_text(texto);
         pausado = 0;
      End
      Frame;
   End
End

Process circulator()
Begin
   circulo(50,100);
   circulo(100,170);
   circulo(150,240);
   circulo(200,310);
   circulo(250,380);
   Loop
      if (key(_s) AND NOT pausado)
           signal(son,s_freeze);
           texto=write(0,320,240,4,"Pausado sólo el último hijo");
           pausado=1;
      end
      if (key(_a) AND pausado)
           signal(son,s_wakeup);
           delete_text(texto);
           pausado=0;
      end
      Frame;
   End
End

Process circulo(x,y)
Private
   Int direc = 10;
end
Begin
   graph = mapCirculo;
   Loop
      x =x+ direc;
      If (x < 50) direc = 10; End
      If (x > 590) direc = -10; End
      Frame;
```

```
   End
End
```

En este caso tenemos un proceso padre, “circulator()”, que genera cinco procesos hijos (y hermanos entre sí), “circulo()”, por lo que enviando desde el programa principal una señal de tipo *_TREE a “circulator()”, controlamos todos los círculos a la vez. Además en este ejemplo, dentro del propio proceso “circulator()”, podemos controlar individualmente el último de los cinco procesos creados por éste mediante la variable local ***SON***

Un caso relativamente frecuente que podemos encontrarnos a la hora de programar videojuegos es querer enviar una señal a todos los procesos sin discriminación (excepto posiblemente el proceso actual o algún tipo de proceso concreto). Es decir,por ejemplo congelar todos los procesos existentes. Vamos a ver una posible manera de hacer esto.

El siguiente código implementa una función (las funciones las estudiaremos unos apartados más adelante) llamada “signal_all()” que nos conseguir lo que pretendemos. He aquí el ejemplo:

```
Import "mod_proc";
Import "mod_video";
Import "mod_map";
Import "mod_text";
Import "mod_key";

Global
   int texto;
   Int pausado = 0;
end

Process Main()
Begin
   set_mode (640,480,32);
   circuloazul(50,100);
   circuloazul(100,170);
   circuloazul(150,240);
   circuloazul(200,310);
   circuloazul(250,380);
   circuloblanco(300,440);
   write(0,0,0,0,"A para pausar todos los procesos");
   write(0,0,10,0,"S para quitar la pausa de todos los procesos");
   write(0,0,20,0,"D para pausar todos los procesos");
   write(0,0,30,0,"F para quitar la pausa de todos los procesos");
   write(0,0,40,0,"ESC para salir");
   Loop
      If (key(_esc)) exit(); End

      If (key(_a) AND NOT pausado)
         signal_all(s_freeze,0);
         texto = write(0,0,50,0,"Pausa total");
         pausado = 1;
      End
      If (key(_s) AND pausado)
         signal_all(s_wakeup,0);
         delete_text(texto);
         pausado = 0;
      End

      If (key(_d) AND NOT pausado)
         signal_all(s_freeze,type circuloblanco);
         texto = write(0,0,50,0,"Pausa total excepto para el cuadrado blanco");
         pausado = 1;
      End
      If (key(_f) AND pausado)
         signal_all(s_wakeup,type circuloblanco);
         delete_text(texto);
         pausado = 0;
      End
```

```
      Frame;
   End
End

Process circuloblanco(x,y)
Private
   Int direc = 10;
end
Begin
   graph = new_map(40,40,32); map_clear(0,graph,rgb(255,255,255));
   Loop
      x =x + direc;
      If (x < 50) direc = 10; End
      If (x > 590) direc = -10; End
      Frame;
   End
End

Process circuloazul(x,y)
Private
   Int direc = 10;
end
Begin
   graph = new_map(40,40,32); map_clear(0,graph,rgb(0,0,255));
   Loop
      x =x + direc;
      If (x < 50) direc = 10; End
      If (x > 590) direc = -10; End
      Frame;
   End
End


/*Esta función envía a todos los procesos una señal concreta. Sus parámetros son: la
señal a enviar (parámetro de tipo int), y el identificador de proceso -o el tipo de
proceso, indicando la construcción "type" en la llamada a la función, tal como hacemos en
este ejemplo-  que se desea que no reciba esa señal (sólo admite un identificador ó tipo
pero es fácilmente modificable). Si no se quiere que haya ninguna excepción, este segundo
parámetro ha de valer 0. Si todo funciona correctamente, esta función retornará 0 (valor
por defecto.*/

Function signal_all(int senal, int procexcepcion)
Private
  int i;
End
Begin
//Envía a todos los procesos de todo tipo una señal, excepto los procesos indicados
for(i=0; i<=65535; i++)
          if(i!=procexcepcion && exists(i))
                    signal(i,senal);
          end
end
End
```

En el código anterior tenemos dos procesos: “circuloblanco()” y “circuloazul()”. De estos últimos existen varias instancias. Como se puede ver, si se pulsa la tecla “A” se envía una señal de congelación a todos los procesos de todos los tipos, y pulsando “S” se descongelan todos ellos. En cambio, si se pulsa “D”, se congelan todos los procesos excepto todos los que sean de tipo “circuloblanco()” (que en concreto sólo hay uno). Pulsando “F” se descongelan todos los procesos anteriormente congelados con “D”.

Podemos estar orgullosos de haber creado nuestra función “signal_all()”, ya que funciona muy bien. No obstante, hemos de saber que hemos reinventado la rueda. La orden **signal()** también es capaz por sí sola de enviar una señal determinada a todos las instancias de procesos existentes: simplemente hay que especificar el valor numérico 0 -o alternativamente, la constante ***ALL_PROCESS*** – como valor de su primer parámetro. Eso sí, haciéndolo de esta manera no podemos especificar excepciones tal como hemos hecho con “signal_all()”.

## *Posibilidad de no acatar señales: orden “Signal_action()”

Hasta el momento, siempre que se ha lanzado una señal mediante **signal()** a una instancia ó tipo de proceso, éste ha hecho caso de ésta y se ha puesto en el estado que le correspondía “sin rechistar”: si la señal decía que el proceso debía congelarse, el proceso se congelaba. No obstante, existe la posibilidad de permitir que el proceso al que va dirigido una señal concreta pueda obviar dicha señal y hacer como si nunca la hubiera recibido. Es decir, podemos hacer que los procesos puedan “no hacer caso” a determinadas señales, en determinados momentos. Esto se logra con la orden **signal_action()**, definida en el módulo “mod_proc”.

Esta orden sirve para establecer cómo reaccionará el proceso especificado como primer parámetro (puede ser una instancia concreta dada por su identificador, un tipo de proceso dada por la expresión “type” o bien la constante ***ALL_PROCESS*** -que es igual a 0-) cuando reciba una señal concreta especificada como segundo parámetro (puede ser cualquiera de las conocidas, incluyendo las *_TREE). La reacción que tendrá ese proceso ante esa señal se establece en el tercer parámetro, y hay dos reacciones posibles: acatar la señal (es el valor por defecto: se especifica con la constante ***S_DFL***, equivalente a 0) ó ignorarla (se especifica con la constante ***S_IGN***, equivalnte a 1).

El primer parámetro de **signal_action()** , (el que señala el proceso al que se le cambiará su reacción ante una señal determinada) es opcional. Si no se especifica, se da por supuesto que se está cambiando el modo de reacción del proceso actual.

Por ejemplo:

```
//El proceso actual ignorará la señal KILL a partir de ahora
signal_action(S_KILL,S_IGN);

/*Todos los procesos EXISTENTES  actualmente (no los que sean creados posteriormente,
ojo!) ignorarán la señal KILL a partir de ahora*/
signal_action(ALL_PROCESS,S_KILL,S_IGN);

/*Todos los procesos existentes actualmente del tipo "Player" (no los que sean creados
posteriormente, ojo!) ignorarán la señal KILL a partir de ahora*/
signal_action(type Player,S_FREEZE,S_IGN);
```

De todas maneras, es posible que en determinadas circunstancias nos interese que un determinado tipo/instancia de algún proceso acate una determinada señal sí o sí a pesar de tener activada la reacción ***S_IGN.***¿Existe alguna manera de “forzar” la ejecución de señales independientemente del estado que establezca **signal_action()**? Sí: utilizando señales “forzosas”. Este tipo de señales especiales siempre se aplicarán a pesar de que el proceso objetivo haya sido programado para ignorarlas. A continuación se presenta una tabla con estas nuevas señales “forzosas”.

| | | |
|---|---|---|
| **S_KILL_FORCE** | Equivale a 50 | Equivalente “forzosa” de **S_KILL** |
| **S_WAKEUP_FORCE** | Equivale a 51 | Equivalente “forzosa” de **S_WAKEUP** |
| **S_SLEEP_FORCE** | Equivale a 52 | Equivalente “forzosa” de **S_SLEEP** |
| **S_FREEZE_FORCE** | Equivale a 53 | Equivalente “forzosa” de **S_FREEZE** |
| **S_KILL_TREE_FORCE** | Equivale a 150 | Equivalente “forzosa” de **S_KILL_TREE** |
| **S_WAKEUP_TREE_FORCE** | Equivale a 151 | Equivalente “forzosa” de **S_WAKEUP_TREE** |
| **S_SLEEP_TREE_FORCE** | Equivale a 152 | Equivalente “forzosa” de **S_SLEEP_TREE** |
| **S_FREEZE_TREE_FORCE** | Equivale a 153 | Equivalente “forzosa” de **S_FREEZE_TREE** |

## *Variables públicas y sentencia “Declare”

Una variable pública es una variable que es específica y única de un proceso/función, igual que lo es una variable privada. Sin embargo, a diferencia de ésta, una variable pública puede ser accedida desde el resto del programa a partir del identificador de ese proceso (*identificadorProceso.nombreVariablePública*) , igual como se accede a una variable local. Es decir, dicho de otra forma, una variable pública sería igual que una variable local, pero sólo existente

para el proceso concreto donde se haya definido, no para todos.

Para declarar variables públicas, haremos uso de la sección PUBLIC/END, de la siguiente manera.

```
PUBLIC
        Declaraciones_y_posibles_inicializaciones;
END
```

A priori, por lógica, la sección anterior debería de escribirse dentro del código del proceso particular al que se le quiere asociar esta variable, al igual que hacemos con las variables privadas, pero enseguida veremos que esto a veces no tiene porqué ser bien bien así: las variables públicas funcionan de una manera algo diferente.

La primera particularidad novedosa de este tipo de variables es que la variable *identificadorProceso* ha de declararse obligatoriamente no como Int -que sería lo normal hasta ahora- sino como un tipo de datos "nuevo" cuyo nombre es igual al nombre de dicho proceso. Es decir, para poder acceder a las variables públicas de un proceso desde otro proceso, el identificador del primero se ha declarar no como int, sino con el nombre del proceso. (En realidad el tipo de dato finalmente acaba siendo un int, pero se ha de declarar tal como se comenta). Prueba primero este código, a ver qué pasa:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";

Global
    Nave ship;
End

Process Main()
Begin
    ship = Nave();
    write(0,100,100,4,ship.name + ":" + ship.speed);
    Repeat
        frame;
    Until(key(_esc))
End

Process Nave()
Public
        int speed = 50;
        String name = "Galactica";
End
Begin
    repeat
        frame;
    until(key(_esc))
End
```

En principio, este código lo único que haría es escribir en pantalla el valor de dos variables locales del proceso "Nave". Fíjate, tal como hemos comentado, que el identificador del proceso no lo declaramos como Int sino con el nombre del proceso al que pertenecen las variables locales que queremos usar: "Nave". No obstante, se produce un error en la compilación (no se reconocen las variables públicas que hemos definido). Nos falta hacer algo más para que las variables públicas estén bien declaradas. Fíjate ahora en este otro código:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";

Global
    Nave ship;
End

Process Nave()
Public
        int speed = 50;
```

```
        String name = "Galactica";
End
Begin
    repeat
        frame;
    until(key(_esc))
End

Process Main()
Begin
    ship = Nave();
    write(0,100,100,4,ship.name + ":" + ship.speed);
    Repeat
        frame;
    Until(key(_esc))
End
```

Si te fijas, es el mismo programa de antes,pero con una diferencia: hemos escrito el código del proceso "Nave()" antes del del proceso "Main()". Esto no lo habíamos explicado hasta ahora, pero es perfectamente posible alterar el orden "habitual" de escritura de los procesos en nuestro código (siempre después, eso sí, de las líneas "import" y las secciones de declaraciones globales y locales, que van al principio de todo siempre). Es decir, aunque hasta ahora habíamos escrito el proceso "Main()" el primero, nada nos impide que sea escrito el último: a Bennu eso no le importa nada.

Si ejecutas este programa, verás que ahora funciona. ¿Por qué? Porque en la primera versión del programa, al llegar a la línea de "Main()" donde se utiliza la variable *ship.name* , el compilador no sabe qué variable es ésta porque se ha declarado en la sección PUBLIC/END de un proceso cuyo código el compilador no ha llegado a leer todavía porque está escrito después del código que está compilando en ese mismo momento, con lo que provoca un error al no saber de dónde sale esa variable. Fíjate que este error no ocurrirá nunca ni con las variables globales ni con las locales, porque éstas se declaran al principio de todo, ni con las privadas, porque éstas sólo se utilizan dentro del código del propio proceso. En cambio, en la segunda versión del programa, al haber escrito el código del proceso "Nave()" antes de "Main()", el compilador ya ha leído previamente el código de "Nave()" y tiene toda la información necesaria para saber reconocer en el momento de la compilación dónde se declararon las variables públicas que se encuentra por su camino. Cuando el compilador llega a leer éstas, él ya sabe qué significan porque ya "repasó" previamente el código de "Nave()", escrito anteriormente a "Main()".

De esta manera ya podemos utilizar las variables públicas sin problemas. Ya está: no hay nada más que aprender de este tema.

No obstante, si no te acabas de acostumbrar al nuevo orden de escritura del código, existe una alternativa. El truco está en mantener el orden del código como siempre (el proceso "Main()" el primero), pero declarar las variables públicas de otra manera. Con la sentencia DECLARE, la cual se escribirá al principio de nuestro código, después de las secciones GLOBAL/END y LOCAL/END si las hubiere, y fuera de cualquier código de proceso. Tiene la siguiente sintaxis:

```
DECLARE PROCESS nombreProceso (tipo param1,tipo param2,...)
    PUBLIC
            Declaraciones_y_posibles_inicializaciones;
    END
END
```

donde "nombreProceso" es el nombre del proceso que albergará las variables públicas declaradas, y los parámetros que se especifiquen serán los mismos que tenga dicho proceso.

Es decir, que la tercera versión de nuestro programa, con la orden DECLARE, quedaría así:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";

Global
    Nave ship;
```

```
End

Declare Process Nave()
          Public
          int speed = 50;
                  String name = "Galactica";
          End
End

Process Main()
Begin
    ship = Nave();
    write(0,100,100,4,ship.name + ":" + ship.speed);
    Repeat
        frame;
    Until(key(_esc))
End


Process Nave()
Begin
    repeat
        frame;
    until(key(_esc))
End
```

Fíjate que cerramos con un END el bloque PUBLIC y con otro END el bloque DECLARE.

Pongo otro ejemplo muy similar al anterior para afianzar estos nuevos conceptos:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";

DECLARE PROCESS mi_proceso()
public
  int hola;
  int adios;
END
END

process main()
begin
          mi_proceso();
          otro_proceso();
end

process mi_proceso()
begin
 hola=20;
 adios=30;
 repeat frame; until(key(_esc))
end

process otro_proceso()
private
/*Se ha de asignar el tipo mi_proceso a la variable "consulta" para poder recoger en ella
el identificador de una instancia cualquiera del proceso "mi_proceso()" y poder acceder
asía sus variables públicas. Si "consulta" se declarara como Int, podría recoger
perfectamente el identificador de una instancia cualquiera del proceso "mi_proceso()",
pero no podría acceder a sus variables públicas. Puedes comprobar esto último: verás que
obtienes un error de compilación*/
 mi_proceso consulta;
end
begin
 consulta=get_id(type mi_proceso);
 write(0,0,0,0,"La var. pública HOLA de 'mi_proceso()' vale: " + consulta.hola);
```

```
 write(0,0,10,0,"La var. pública ADIOS de 'mi_proceso()' vale: " + consulta.adios);
 repeat frame; until(key(_esc))
end
```

En este ejemplo, declaro dos variables públicas llamadas *hola* y *adios* para el proceso "mi_proceso()", asignándoles de paso un valor inicial. Y desde otro proceso diferente, accedo al valor de estas variables, a partir del identificador del proceso "mi_proceso()" obtenido gracias a la función **get_id()**, y lo muestro por pantalla. Fíjate (repito otra vez) que la manera de acceder al contenido de una variable pública desde otro proceso es idéntica a la manera de acceder al de una variable local, con la salvedad de que el identificador del proceso que tiene definida la variable pública a la que se quiere acceder ha de ser declarado no como Int sino con el nombre de dicho proceso, tal como se puede observar también en el ejemplo.

Hablando más en general, la sentencia DECLARE, como su nombre indica, realmente sirve para declarar no ya variables, sino un proceso (o una función: en seguida veremos qué es lo que son) antes de utilizarlo en el código del programa. Es decir, DECLARE permite definir procesos y funciones antes de que vayan a ser implementados. Esto es útil, como hemos visto, cuando el compilador necesita saber de la existencia de ese proceso antes de que el código de éste lo haya leído.

La aplicación más habitual es la que hemos visto: declarar variables públicas de ese proceso en concreto, pero también es posible incluir dentro del bloque DECLARE/END un bloque PRIVATE/END para declarar también las variables privadas que utilizará el proceso en cuestión. No obstante, hay que tener en cuenta que si se declaran variables privadas en un bloque DECLARE, no se podrá declarar después ninguna otra variable privada dentro del código específico de ese proceso: todas las variables privadas deberán ser declaradas dentro del bloque DECLARE.

## *Las órdenes "Let_me_alone()" y "Exit()"

Como bien sabes, nuestros programas constan de un código principal, a partir del cual se van llamando a diferentes procesos,y éstos a otros, etc. A partir de ahí, cada proceso "es como si" fuera un programa independiente que está ejecutándose en paralelo. Esto quiere decir que, por ejemplo, si hacemos que el proceso principal termine en pulsar la tecla ESC, esto no quiere decir ni mucho menos que nuestro programa de deje de ejecutar, ya que posiblemente tendremos multitud de procesos continuando su ejecución gracias a sus bucles loop infinitos. Es probable que hayas notado esta circunstancia en alguno de los ejemplos anteriores cuando intentaste acabar la ejecución de un programa y aparentemente éste no respondía: el código principal sí que había terminado pero el código de los diferentes procesos todavía estaba funcionando, a perpetuidad.

Para solucionar éste y otros posibles problemas que pudieran surgir, tenemos la función **let_me_alone()** -"déjame solo" literalmente-. **Let_me_alone()** acaba con todos los procesos existentes SALVO el que ejecuta esta instrucción. Si por ejemplo colocamos esta función justo al final del código principal, nos aseguramos de matar todos los procesos primero, para seguidamente acabar la ejecución del programa de forma limpia, sin tener que controlar cuántos procesos activos hay. Esta solución es útil también cuando se quiere salir del juego y volver a alguna especie de menú, introducción o parecido.

Un ejemplo que creo que no necesita demasiada explicación:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";
Import "mod_map";
Import "mod_video";

global
  int bolas=1;
end

process main()
begin
  set_mode(640,480);
  write(0,260,30,4,"SPACE = elimina todos los procesos excepto el programa principal");
  write(0,260,50,4,"ENTER = crea de nuevo los procesos gráficos");
  bola(260,160);
```

```
  bola(300,200);
  bola(260,240);
  bola(220,200);
  repeat
/*La condición de "bolas==1" es para que sólo se realice el interior del if si los
gráficos son visibles en ese momento*/
    if(key(_space) && bolas==1)
      let_me_alone();
      bolas=0;
    end
/*La condición de "bolas==0" es para que sólo se realice el interior del if si los
gráficos no son visibles en ese momento*/
    if(key(_enter) && bolas==0)
      bola(260,160);
      bola(300,200);
      bola(260,240);
      bola(220,200);
      bolas=1;
    end
    frame;
  until(key(_esc))
  exit();
end

process Bola(x,y)
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(0,255,0));
  loop
    frame;
  end
end
```

La función **let_me_alone()** es parecida a **exit()**, pero **exit()** -ya lo habrás visto en algún ejemplo anterior- acaba con TODOS los procesos existentes y por tanto, finaliza el juego, sin esperar ni siquiera al siguiente frame.

La orden **exit()** tiene dos parámetros opcionales: el primero es una cadena de texto que se mostrará por consola al acabar el programa. El segundo será un número entero que se devolverá al sistema operativo, para que éste sepa en qué estado ha terminado el programa; si no se especifica ningún código numérico, por defecto es 0 e indica que el programa terminó correctamente. Si se especifica cualquier otro valor, generalmente indicará que se ha terminado de forma abrupta, no esperada ó incorrecta.

También has de saber que existe una variable global predefinida interesante llamada ***EXIT_STATUS,*** que pasa a valer 1 cuando el usuario trata de cerrar la aplicación desde el sistema operativo (por ejemplo, si el juego se ejecuta en una ventana, cuando pulsa el botón de cerrar de su barra de título). Su valor normal y original es 0. Para que un juego funcione correctamente y esté bien integrado en el sistema operativo, se recomienda consultar esta variable con un proceso y salir del juego (por ejemplo mediante la función **exit()** sólo cuando esta variable valga 1. La consulta debe realizarse cada frame , pues no se garantiza que la variable siga valiendo 1 en frames posteriores al intento de cerrar la aplicación.

## *La orden "Exists()"

Otro comando interesante relacionado con el reconocimiento y detección de procesos activos durante la ejecución del programa es el comando **Exists().** Esta función devuelve 1 (cierto) si el proceso cuyo identificador recibe como primer y único parámetro sigue en ejecución (esto es, no ha sido eliminado de memoria, ni marcado para matar mediante una función como **signal()**), y devuelve 0 (falso) si no hay ni rastro de ese proceso. Es posible también pasarle como parámetro un tipo de proceso obtenido mediante la instrucción TYPE, en cuyo caso devolverá 1 si hay en memoria al menos un proceso del tipo indicado. En caso de no encontrar ningún proceso, EXISTS devolverá 0.

Un ejemplo de uso de este comando podría ser el siguiente:

```
Import "mod_proc";
```

```
Import "mod_text";
Import "mod_key";
Import "mod_map";
Import "mod_video";

global
  int IDB;
  int ex;
  int ebola=1;
end

process main()
begin
//  set_mode(640,480,32);
  write(0,160,30,4,"Proceso Bola Existente...");
  write_var(0,160,40,4,ex);
  write(0,160,60,4,"SPACE = Elimina el proceso 'Bola'");
  write(0,160,80,4,"ENTER = Vuelve a crear el proceso 'Bola'");
  IDB=Bola();
  repeat
    ex=exists(IDB);
/*La condición de "ebola==1" es para que sólo se realice el interior del if si el proceso
"Bola" existe en ese momento*/
    if(key(_space) && ebola==1)
      signal(IDB,s_kill);
      ebola=0;
    end
/*La condición de "ebola==0" es para que sólo se realice el interior del if si el proceso
"Bola" no existe en ese momento*/
    if(key(_enter) && ebola==0)
      IDB=Bola();
      ebola=1;
    end
    frame;
  until(key(_esc))
/*La línea siguiente elimina todas las demás instancias de procesos que puedan existir
excepto ésta,(en realidad, sólo existe una instancia del proceso bola y nada más).Al ser
la última línea del único proceso que queda en pie, cuando se acabe éste, se finalizará
completamente el programa*/
  let_me_alone();
end

process Bola()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
  x=160;
  y=110;
  loop
    frame;
  end
end
```

## *La cláusula "OnExit"

Hay casos donde el programador asigna recursos a un determinado proceso ( impresion de un texto, carga de un sonido o un fpg, creacion de una tabla blendop,etc) y en un momento dado ese proceso es "killeado" por otro proceso diferente -por ejemplo: cuando un enemigo es matado por nuestra nave-. En ese momento, cuando ese proceso está a punto de morir,es posible que nos interese realizar la descarga de los recursos que estaba utilizando, porque se dé el caso que éstos ya nos los utilizará ningún proceso más. El hecho de que un proceso muera no implica que los recursos que esté utilizando sean descargados de la memoria, ni mucho menos, (con el desperdicio de memoria y el ralentizamiento del juego que esto comporta). Para facilitar esta labor, disponemos de la cláusula ONEXIT:

Su sintaxis es la siguiente:

```
Process ... //Cabecera del proceso
//Sección de declaraciones de variables privadas/públicas (si es necesario)
Begin
//Código del proceso
OnExit
//Código de salida (normalmente, para descargar recursos utilizados por el proceso)
End
```

Por lo que puedes ver, la cláusula OnExit se usa al final de un bloque BEGIN/END de un proceso. Y el código que hay posterior a ella se ejecutará cuando el proceso/función termine su ejecución, ya sea de "muerte natural" al llegar al final de su código o bien porque haya recibido una señal Kill.

En realidad, después de la cláusula OnExit se puede poner cualquier código que se te ocurra, pero ya he comentado que su utilidad más evidente es escribir código de descarga de recursos. Piensa, por ejemplo, que si se escribe después de la cláusula OnExit algo como *Loop frame;End,* el juego se quedará bloqueado, ya que habrá un proceso que no terminará de morir nunca y se quedará en un estado inestable.

Un ejemplo (necesitarás un FPG llamado "graficos.fpg" con una imagen en su interior de código 001):

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";
Import "mod_map";
Import "mod_video";

global
            int idfpg;
            int idproc;
end

process main()
Begin
    set_mode(640,480,32);
    idfpg=load_fpg("graficos.fpg");
    idproc=proceso1();
    Repeat
        if(key(_x)) signal(idproc,s_kill); end
        frame;
    Until(key(_esc))
End

Process proceso1()
Private
    int idtexto;
End
Begin
    file=idfpg;
    graph=1;x=320;y=240;
    idtexto=write(0,100,100,4,"Hola");
    Loop
          frame;
    End
OnExit
    unload_fpg(file);
    delete_text(idtexto);
End
```

Otro ejemplo más ilustrativo, si cabe:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";
Import "mod_map";
```

```
process main()
private
   int i;
end
begin
   i= test_process();
   loop
      if (key(_e))
        if (exists(i)!=0) signal(i,s_kill); end
      end
      if (key(_esc)) exit(); end
      frame;
   end
end

process test_process()
begin
   write(0,0,0,0,"Proceso creado - Pulsa (E) para matarlo");
   loop
     frame;
   end
OnExit
   write(0,0,10,0,"Proceso matado.");
end
```

Un detalle importante a saber es que un proceso cuando pasa a ejecutar el código del interior de la cláusula OnExit, pasa a estar en un estado “DEAD”, y sólo al final, cuando ya no tenga ningún código más que ejecutar, pasará a estar “KILLED”.

## *La cláusula CLONE

Esta cláusula puede estar escrita en cualquier punto del interior del código de un proceso. Su sintaxis es así:

```
CLONE
    Sentencia1;
    Sentencia2;
    ...
END
```

Esta sentencia (palabra clave del lenguaje) crea un nuevo proceso idéntico al actual, con la salvedad de que las sentencias entre las palabras reservadas CLONE y END se ejecutarán únicamente en el nuevo proceso y no en el actual. Por ejemplo, si cualquier proceso del programa, con unas coordenadas (***X, Y***) determinadas y un gráfico (***GRAPH***) concreto, ejecuta la siguiente sentencia:

```
CLONE
 x=x+100;
END
```

lo que pasará es que se creará un nuevo proceso identico a él, con el mismo gráfico y los mismos valores en todas sus variables, a excepción de la coordenada x, que el nuevo proceso tendrá 100 puntos más a la derecha.

Esta sentencia se utiliza para crear réplicas de un proceso, dividirlo en dos procesos (casi) iguales.

Un ejemplo donde se ve mejor todo esto:

```
PROCESS Main()
BEGIN
    // ...
    x=0;
    y=0;
    CLONE
```

```
        x=x+10;
    END
    CLONE
        y=y+10;
    END
    // ...
END
```

En este ejemplo, las 2 sentencias CLONE crearán 3 copias del proceso principal (y no 2, como podría haberse esperado). Al ejecutarse la primera sentencia CLONE se creará un nuevo proceso, con lo que habrá 2: uno en (x=0, y=0) y otro en (x=10, y=0). Y estos dos procesos ejecutarán la segunda sentencia CLONE, el primero (el original) creando con ello un nuevo proceso en (x=0, y=10), y el segundo creará el nuevo proceso en (x=10, y=10). Para crearse únicamente 2 copias del proceso original se podría haber construido el programa, por ejemplo, de la siguiente forma:

```
PROCESS Main()
BEGIN
    // ...
    x=0;
    y=0;
    CLONE
        x=x+10;
        CLONE
            y=y+10;
        END
    END
    // ...
END
```

Se puede ver ahora que el proceso original (x=0, y=0) creará uno en (x=10, y=0) y éste, a su vez, otro en (x=10, y=10), creandose únicamente dos copias del original. Se debe, por tanto, tener mucho cuidado con el uso de la sentencia CLONE de forma secuencial o dentro de un bucle, pues se debe contar con que los primeros 'clones' pueden crear, a su vez, a nuevos 'clones'.

Un último ejemplo, completamente funcional para probar:

```
import "mod_key";
import "mod_map";
import "mod_video";
import "mod_proc";

Process Main()
Begin
    set_mode(320,240,32);
    squares();
    repeat
        frame;
    until(key(_ESC))
    exit();
End

Process squares()
Private
   int advance;
End
Begin
    graph = new_map(5,5,32);
    y=120;
    map_clear(0,graph,rgb(255,0,255));
    advance = 1;
    clone
        graph = map_clone( 0, graph );
        map_clear(0,graph,rgb(255,255,255));
        advance = 2;
    end
```

```
    loop
        x = x +advance;
        frame;
    end
onexit:
    unload_map(0,graph);
End
```

## *Sobre la ejecución paralela de procesos. La variable local predefinida PRIORITY

Ejecuta el siguiente código:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";
Import "mod_map";

process main()
begin
          proceso1();
          proceso2();
          loop
                    if(key(_esc)) exit();end
                    frame;
          end
end

process proceso1()
begin
  x=320;
  loop
    if(key(_a)) x=x+1; end
    delete_text(0);
    write(0,20,10,4,x);
    frame;
  end
end

process proceso2()
begin
 x=220;
 loop
    if(key(_b)) x=x+1; end
    delete_text(0);
    write(0,50,10,4,x);
    frame;
  end
end
```

Miremos qué hace este programa con detenimiento. Vemos que el código principal básicamente llama *PRIMERO* a "proceso1()" y *DESPUÉS* a "proceso2()". Tanto "proceso1()" como "proceso2()" son "invisibles" porque no hemos asociado ningún gráfico con la variable GRAPH, pero eso no quita que estén funcionando como cualquier otro proceso. En concreto, a ambos le asignamos una coordenada X diferente. Y a partir de entonces, en ambos entramos en el bucle Loop. En "proceso1()" lo que hacemos indefinidamente es borrar todos los textos de la pantalla, escribir el valor actual de su X y plasmarlo en un fotograma (con la opción previa de haber modificado su valor pulsando una tecla). En "proceso2()" hacemos lo mismo: borramos todos los textos de la pantalla, escribimos el valor actual de su X y lo plasmamos en el fotograma. Un rápido (y erróneo) análisis de este código puede llevar a pensar que lo que se tendría que imprimir en pantalla son los dos valores de cada una de las X de los dos procesos. Pero no. Sólo se imprime la X de "proceso2()".¿Por qué?

He explicado en apartados anteriores que todos los procesos muestran en pantalla de forma sincronizada el resultado de sus líneas de código anteriores a cada línea frame que se encuentren en su ejecución.De forma

sincronizada quiere decir que hasta que todos los procesos activos existentes no hayan llegado a una línea frame el resto de procesos esperará en su línea frame correspondiente: y cuando por fin todos los procesos hayan llegado a ella, es cuando el fotograma se muestra por pantalla ofreciendo a la vez el resultado de todos los cálculos efectuados en las líneas anteriores de cada proceso.

Pero hay que tener cuidado. Siguiendo esta interpretación, uno podría pensar del código anterior lo siguiente: pongo en marcha los dos procesos, en uno se borra toda la pantalla, se escribe su X y se llega al frame y en el otro se borra la pantalla, se escribe su X y se llega al frame, con lo que cuando llegan al frame los dos procesos, uno ha de escribir su X y el otro la suya. ¡NO!Porque el código de los procesos se ejecuta se forma secuencial: un código de proceso después de otro. Es decir, el primer proceso que sea creado desde el programa principal ejecutará primero sus líneas de código hasta llegar a su frame, momento en el que se para; el proceso que haya sido creado justo después del anterior es en ese momento cuando comenzará la ejecución de su código, hasta llegar a su frame, y así con todos los procesos hasta que todos hayan llegado al frame, momento en el que, entonces sí que sincronizadamente, se visualiza el resultado en pantalla.

Esto implica, en el ejemplo anterior, que primero se ejecutan las líneas de "proceso1()" hasta el frame, y luego las líneas de "proceso2()" hasta el frame, y es en ese momento cuando (como ya no quedan más procesos más) se visualiza el resultado. Así, lo primero que ocurre es que se borran todos los textos de la pantalla, luego se escribe el valor de X de "proceso1()", luego se vuelve a borrar toda la pantalla, luego se escribe el valor de X de "proceso2()" y es en ese instante cuando se muestra en pantalla el resultado de todo esto. Puedes entender ahora perfectamente ahora por qué sólo se ve la X de "proceso2()": porque el *delete_text(0)* que aparece en "proceso2()" borra todos los textos anteriormente escritos, que en este caso es la X de "proceso1()".

Si cambias el orden de invocación de los procesos en el programa principal (es decir, llamas primero a "proceso2()" y luego a "proceso1()"), verás que ahora el resultado es al revés: sólo se ve en pantalla la X de "proceso1()", por el mismo motivo. Nota: evidentemente, existe una manera más óptima de visualizar el valor cambiante de una variable que la mostrada en el ejemplo anterior: escribiendo un **write_var()** fuera del loop de los procesos. Pero el código se ha escrito como excusa para demostrar la explicación dada de sincronía y secuencialidad.

Después de toda la explicación precedente, ya estamos en condiciones de entender que la orden Frame no dibuja nada, sino que lo que realmente es ceder la ejecución al próximo proceso. Aunque aparentemente Bennu ejecute los procesos de forma concurrente, realmente el procesador sólo los puede atender de uno en uno, y para que se pase al siguiente proceso a ejecutar, o se termina con el proceso actual o se lee la instrucción Frame. Solamente cuando todos los procesos hayan terminado de ejecutarse o han llegado a su Frame respectivo, se dibujan en pantalla, se espera el tiempo que sobre (ver el comando **set_fps()**) y se pasa al siguiente Frame.

Seguramente te estarás preguntando ahora si no tenemos más remedio que limitarnos a respetar el orden de ejecución impuesto por el orden de creación de los procesos. ¿No hay ninguna manera de hacer que el código de un proceso se ejecute antes que el de otro, aunque éste haya sido creado antes que aquél? Pues sí, hay una manera: utilizando la variable local predefinida ***PRIORITY***, la cual define el nivel de prioridad del proceso pertinente.Por defecto su valor es 0.

Los procesos que tienen una prioridad mayor se ejecutan antes ya que a la hora de dibujar cada FRAME del juego lo primero que hace el intérprete es utilizar el valor de esta variable para determinar el orden de ejecución de los procesos. Recuerda que los procesos se ejecutan primero y sólo cuando ya no queda ningun proceso por ejecutar en el FRAME actual, se dibujan en pantalla.

Para que veas en acción esta variable, jugaremos con el ejemplo siguiente:

```
Import "mod_proc";
Import "mod_text";
Import "mod_key";
Import "mod_map";

process main()
begin
  proceso1();
  loop
          if(key(_esc)) exit();end
          frame;
  end
end
```

```
process proceso1()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
  x=160;y=120;
  proceso2();
 loop
            if(key(_left)) x=x-10;end
            if(key(_down))y=y+10;end
            if(key(_right))x=x+10;end
            if(key(_up))y=y-10;end
            frame;
  end
end

process proceso2()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,255));
  x=160;y=120;
  priority=1;
  loop
            x=father.x;
            y=father.y;
            frame;
  end
end
```

En este ejemplo tenemos un proceso “proceso1()” -un cuadrado blanco- que podremos mover con el cursor, y además es padre de “proceso2()” -un cuadrado rosa-, el cual, lo único que hace seguir en cada fotograma la posición de “proceso1()”. Lo importante del asunto es que te fijes que, cuando pulsas los cursores, parece que el gráfico de “proceso1()” se avanza un poquito al de “proceso2()”, dejando entrever éste; aunque cuando dejamos de pulsar los cursores ambos gráficos vuelven a superponerse perfectamente. ¿Por qué es esto? Porque el código de “proceso2()” es el que se ejecuta primero hasta la línea Frame (es el que tiene prioridad mayor, ya que si no se especifica nada, las prioridades de los procesos son igual a 0, y 1>0), después se ejecuta el código de “proceso1()” hasta el Frame y cuando todos los procesos han llegado a su Frame, se pinta por pantalla el resultado. Estudiemos esto con más detenimiento.

Está claro que “proceso2()” tomará la X e Y de la posición que en este momento tiene su padre (“proceso1()”), la cual puede cambiar si se está pulsando los cursores. Pero “proceso1()” se ejecuta después que “proceso2()”, con lo que en cada frame, la X e Y de “proceso2()” será la X e Y que tenía “proceso1()” en el frame anterior, por lo que “proceso2()” siempre irá retrasado un frame respecto a “proceso1()”, y es por eso el efecto. Es decir, que “proceso2()” no se enterará de que su padre ha cambiado su posición hasta que no pase el siguiente frame, porque el código de su padre siempre se ejecuta después de el suyo propio, con lo que los movimientos que pueda sufrir el padre ya nos notará hasta que se vuelva a reiniciar una vuelta entera de ejecuciones para el nuevo frame. Parece un poco lioso pero si lo piensas un poco verás que tiene su lógica.

¿Qué pasaría si la línea priority=1, en vez de estar en el código de “proceso2()” estuviera en el de “proceso1()”? Haz la prueba. Estaremos haciendo que la prioridad de “proceso1()” sea mayor que la de “proceso2()” (1>0) y por lo tanto, el código de “proceso1()” se ejecutará antes que el de “proceso2()”. Una vez que ambos códigos hayan llegado a su orden FRAME respectiva -junto con la del proceso principal-, es cuando se mostrará el resultado por pantalla. Así pues, la cosa cambia. Estudiemos esto con más detenimiento.

Si pulsamos una tecla del cursor, cambiamos la X e Y de “proceso1()”, y llegamos al Frame. Entonces pasaremos la ejecución al código de “proceso2()”, en el cual se toman los valores actuales de X e Y de “proceso1” y se asignan a la X e Y de “proceso2()”, y llegamos al Frame. Una vez ya hemos llegado al Frame de todos los procesos, es cuando se pinta la pantalla: pero fíjate que en este momento la X e Y de “proceso1()” y de “proceso2()” siempre valen exactamente lo mismo, por lo que veremos siempre sólo el cuadrado blanco, ya que ambos procesos se mueven a la vez y el gráfico de uno tapará al otro. ¿Por qué es el cuadrado blanco el que tapa al rosa y no al revés? Esto todavía no lo hemos estudiado:es debido a la profundidad de dibujado de los procesos regido por el comportamiento de una variable local que ya veremos: ***Z***.

## *El parámetro de la orden Frame

En el capítulo 2 ya se vió algun ejemplo de uso del parámetro -opcional- de la orden Frame;, pero ya que posiblemente éste es uno de los conceptos que más cuesta de entender al iniciado en Bennu, no está de más mostrar un ejemplo específico. Es un código bastante simple con el que confío se acabe de entender la utilidad de este parámetro.

El parámetro del FRAME lo que indica es cuántos fotogramas más han de pasar extra para que la orden frame sea realmente efectiva. Es decir, si no ponemos nigún parámetro, o ponemos el valor 100 –es lo mismo-, cada vez que se llegue a la línea FRAME, el fotograma de ese proceso está listo para ser imprimido de ipsofacto (a falta de esperar que los otros posibles procesos lleguen también a sus respectivas líneas FRAME). Si se pone otro número como parámetro del FRAME, por ejemplo el 500, lo que ocurrirá es que cuando se llegue a la línea FRAME, el fotograma del proceso no estará listo inmediatamente sino que ese proceso esperará a que los demás procesos hayan ejecutado 5 veces una línea FRAME para ejecutar él la suya (es decir, esperará a que se hayan visualizado 5 fotogramas por parte de los demás procesos antes se visualizar nada él). Es como si el programa se quedara esperando un rato más largo en la línea FRAME, hasta ejecutarla y continuar.

Así pues, este parámetro permite, si se da un valor mayor de 100, ralentizar las impresiones por pantalla de un proceso determinado. También es posible poner un número menor de 100. En ese caso, lo que estaríamos haciendo es "abrir la puerta" más de una vez en cada fotograma: si pusiéramos por ejemplo 50, estaríamos doblando la velocidad normal de impresión por pantalla. Puedes comprobarlo con el siguiente que espero que ayude a entender este concepto:

```
Import "mod_text";
Import "mod_key";
Import "mod_video";
Import "mod_proc";

Process Main()
Private
    int a;
End
Begin
    set_fps(5,0);
    proceso();
    write_var(0,0,0,0,a);
    repeat
        a++;
        frame(500);
    until(key(_esc))
    exit();
End

Process proceso()
Private
    int a;
End
begin
    write_var(0,0,10,0,a);
    loop
         a++;
         frame;
    end
end
```

Al ejecutar el código anterior, verás el incremento indefinido del valor de la variable privada *a* del proceso principal y el incremento del valor de la variable privada *a* del proceso "proceso()", pero a una velocidad diferente, aproximadamente de 5 a 1. Esto es debido a que por cada fotograma de "Main()", han sido mostrado cinco de "proceso()". En el proceso principal se produce una "pausa" que dura cinco fotogramas en su línea frame, que evita que el bucle pueda seguir ejecutándose de forma inmediata y por tanto, el incremento de la variable se produce mucho más espaciadamente en el tiempo.

## *Concepto y creación de una función

Una función es un comando que ejecuta un cierto código y que al finalizar nos devolverá un resultado (de cualquier tipo: entero, decimal, cadena...),el cual podremos recoger, o no, según nos interese. Una función es una manera para "encapsular" con un nombre una porción de código fuente de forma que cuando sea necesario éste, y las veces que sea necesario, no haya que escribirlo siempre integramente sino que simplemente se llame (se "invoque") por su nombre y entonces su código interno –escrito solamente una vez- se ejecutará. Utilizar funciones es una forma de trabajar mucho más elegante, eficiente y fácil de depurar.

Hemos visto muchas funciones hasta ahora: **load_png(), key(), let_me_alone()**... Estas funciones son pequeños programas en sí mismos que se ejecutan siempre que las llamamos: a nosotros no nos interesa cómo se han escrito internamente estas funciones: lo que nos interesa es que realicen una determinada tarea, devolviendo o no un valor -en el caso de las funciones anterior, de tipo numérico entero- que podremos utilizar posteriormente para lo que consideremos oportuno.

Pues bien, nosotros también podemos crear nuestras propias funciones en Bennu. Pero, ¿qué diferencia hay entre crear una función y crear un proceso? Cuando creamos un proceso, este nuevo proceso se pondrá en marcha a la vez que continúa el proceso padre, de manera que los dos procesos se ejecutarán de forma paralela, (concurrente). Cada uno de estos procesos tendrá el comando Frame; y cada uno de ellos llegará por su cuenta a este comando sincronizándose en ese momento y ejecutándose pues a la vez en cada fotograma. Una función es diferente: una función es un trozo de código igual que un proceso, pero cuando una función es llamada, el proceso que la llamó se detiene automáticamente (se "congela") hasta que la función acabe de hacer lo que tiene encomendado. En ese instante, el proceso que la llamó se "despierta" automáticamente y continúa la ejecución de su propio código. Por lo tanto, las funciones NUNCA tienen la orden Frame;, porque no son más que trozos de código que se ejecutan de forma secuencial y ya está, parando mientras tanto la ejecución del proceso que lo llamó. Fíjate: por ejemplo, la función **load_png()** (y todas las demás) hacen esto: hasta que no se haya cargado una imagen PNG el código del proceso no continúa.

Acabamos de decir que mientras se está ejecutando el código del interior de una función, el proceso que la ha llamado se queda en espera de que la función termine -y opcionalmente, devuelva algún valor-, para continuar en la siguiente de su propio código. En principio, pues, el llamar a una función sólo "paraliza" a un proceso, pero fíjate que como en Bennu hasta que todos los procesos no han llegado a una orden Frame; no se pasa al siguiente fotograma, si existe un proceso que se ralentiza por el uso de una determinada función -es posible que ocurra-, esto provocará el ralentizamiento general del juego ya que los demás procesos están esperando a aquél.Lo acabado de comentar implica algo muy importante: las funciones no pueden contener bucles infinitos bajo ningún concepto. La razón es evidente: si una función tuviera un bucle infinito, nunca finalizaría su ejecución y por tanto todos los procesos del programa permanecerían congelados en espera un tiempo infinito, con lo que tendríamos nuestro programa "colgado".

Que las funciones nunca incorporan en su código la orden Frame no es cierta del todo. Pueden incorporarla, pero entonces gran parte (aunque no todas) de sus características se asimilan a la de un proceso normal y corriente, por lo que la diferencia entre ambos se hace más ténue y no resultan tan útiles. En este curso no veremos ninguna función con Frame.

Así pues, ¿qué hay que escribir para crear una función? Las funciones se crean igual que un proceso cualquiera, como los que hemos creado hasta ahora, pero tienen dos o tres pequeñas grandes diferencias:

| |
|---|
| **En vez de escribir la palabra PROCESS en la cabecera de nuestra función, escribiremos la palabra FUNCTION.** Opcionalmente, igual que los procesos, puede recibir parámetros. En ese caso, la regla de escritura de éstos en la cabecera son las mismas que para los procesos. |
| **En el caso de que la función devuelva un valor que no sea de tipo INT , habrá que indicarlo explícitamente en su cabecera, inmediatamente después de la palabra FUNCTION.** Por ejemplo, si una función llamada por ejemplo "hola()", sin parámetros, devuelve un valor decimal, su cabecera será así: *FUNCTION float hola()* |
| **Nuestra función NO ha de tener ningún comando FRAME ni ningún bucle infinito** (ya hemos comentado este aspecto). |
| **Nuestra función podrá devolver opcionalmente un valor. Para ello, se utilizará el comando *RETURN(valor);*.**Pero cuidado porque **con RETURN se acaba la ejecución de la función**, allí donde esté, por lo que hay que saberla colocar en el lugar adecuado. |

Lo que hace esta nueva sentencia que acabamos de conocer, RETURN**,** (la cual en realidad es una palabra reservada del lenguaje y por tanto no está definida en ningún módulo) es finalizar –"matar"- la función actual de forma inmediata, como si se llegara al END de su BEGIN. Se pone entre paréntesis un valor –escrito explícitamente o bien a través de una variable que lo contenga- que queremos devolver al proceso que llamó a dicha función (el cual recogerá ese valor en alguna variable suya, seguramente). Es decir, escribir *return(29);* hará que la función finalice su ejecución y devuelva al proceso invocante el valor 29; y ese proceso recogerá ese valor y ya hará con él lo que estime oportuno. Pero el valor escrito entre los paréntesis del RETURN es opcional. También no se puede escribir nada, dejando los paréntesis vacíos (o incluso, que es lo mismo,sin escribir ni siquiera los paréntesis: sólo *RETURN;*). En este caso, la función termina su ejecución igual que en la otra manera, pero ahora el valor de retorno de la función -que no se suele utilizar demasiado en realidad- será el código identificador de ésta (ya que, las funciones al igual que los procesos tienen cada una un código identificador). Si la función carece de sentencia RETURN (que es perfectamente posible), cuando llegue al final de su código -el último END- también devolverá el código identificador de dicha función al proceso que lo llamó.

La sentencia RETURN de hecho también se puede utilizar dentro del código perteneciente a un proceso, aunque sólo en su forma más simple (es decir, sin escribir ningún valor dentro de los paréntesis), así *RETURN;* Esta sentencia dentro de un proceso tendrá el mismo efecto que dentro de una función: finaliza inmediatamente la ejecución de su código (y por tanto, en el caso de los procesos, destruye esa instancia concreta de proceso de la memoria). Si escribes la sentencia RETURN en el código principal lo finalizará, pero recuerda que si quedan procesos vivos éstos se seguirán ejecutando.

Es evidente que, al no ser un proceso, las funciones no tienen definidas ninguna de las variables locales predefinidas de cualquier proceso: así que ***GRAPH,X,ANGLE***…no tendrán ningún sentido para una FUNCTION.

Posiblemente, después de haber leído los párrafos anteriores tan técnicos no creo que hayas entendido muy bien todavía cómo funcionan las funciones. Básicamente lo que tienes que recordar es que para escribir una función (que no deja de ser una porción de código que no se visualiza en pantalla pero que realiza algún determinado cálculo) se ha de hacer como si escribiéramos un proceso normal, pero éste NO ha de tener la orden FRAME, la cabecera ha de poner FUNCTION en vez de PROCESS seguido del tipo de datos devuelto (si éste no es INT) y si se pretende que la función retorne algún resultado, ha de tener la orden RETURN(*valor*), o simplemente RETURN si se quiere salir del código de la función en algún momento antes de su END.

Un ejemplo, que espero que lo aclare:

```
Import "mod_text";
Import "mod_rand";
Import "mod_key";

Process Main()
private
          int c;
end
begin
          loop
                    c=mifuncion(rand(2,8));
                    write_var(0,100,100,4,c);
                    if(key(_esc)) break; end
                    frame;
          end
end

function mifuncion(int a)
private
          int b;
end
begin
          b=a*a;
          return (b);
end
```

Lo que hemos hecho aquí es crear un nuevo comando o función llamada "mifunción()" que recibe un parámetro entero. El objetivo de esta función es realizar un cálculo y devolver el resultado entero (en este caso, la multiplicación por sí mismo del valor pasado por parámetro, el cual era un número aleatorio). Fíjate que en el programa

principal se recoge lo que devuelve “mifuncion()” en otra variable y a partir de aquí se continúa ejecutando el programa principal.

La gracia de este ejemplo es ver que el programa principal no continúa hasta que “mifuncion()” haya acabado (se deja de ejecutar el código del programa principal para comenzarse a ejecutar el código de la función “mifuncion()”, hasta que éste acabe), y que, como siempre, la sincronización con otros procesos (si el proceso principal se ha retrasado un pelín respecto los demás) se realizará en su frame; correspondiente.

Por experiencia, aconsejo que si se quiere que una función devuelva algún valor, éste se guarde en una variable privada de la función, que será la que se coloque entre paréntesis del RETURN, tal como he hecho en el ejemplo. Como ves, no hay ninguna necesidad de nombrar a las diferentes variables involucradas (“a”,”b”,”c” en el ejemplo) de la misma manera, porque cada una de ellas son variables diferentes del resto.

A continuación presento un código rebuscado a propósito: no tiene ninguna utilidad práctica pero sirve para comprobar si se ha entendido todo lo explicado hasta ahora. Antes de ejecutar nada: ¿qué es lo que saldrá por pantalla?

```
Import "mod_text";

global
          int v=2;
          int w;
end

process main()
begin
          p();
end

function p()
begin
          v=3;
          q();
end

function q()
begin
          w=v;
          r();
end

process r()
begin
          write(0,1,1,0,w);
          loop
                    frame;
          end
end
```

Las funciones pueden ser recursivas: no existe ningún inconveniente (de hecho, los procesos también). Una función recursiva es la que dentro de su código se llama a sí misma. ¿Sabrías razonar por qué se ve un “10” en la pantalla si ejecutas el código siguiente?

```
Import "mod_text";

process main()
private
          int resultado;
end
begin
  resultado=prueba(4);
  write(0,100,100,4,resultado);
  loop frame; end
end

function prueba(int a)
```

```
begin
if(a < 10)
   a = prueba(a+1);
end
return(a);
end
```

A continuación presento un ejemplo de funciones que devuelven diferentes tipos de datos:

```
Import "mod_text";
Import "mod_key";
Import "mod_proc";

global
          int i;
          byte b;
          word w;
          dword dw;
          float f;
          string c;
end

process main()
begin
write(0,150,60,4,cero());
write(0,150,70,4,uno());
write(0,150,80,4,dos());
write(0,150,90,4,tres());
write(0,150,100,4,cuatro());
write(0,150,110,4,cinco());
write(0,150,120,4,seis());
repeat frame; until(key(_esc))
end

//Una función típica de las de toda la vida
function cero()
begin
i = 0;return i;
end

//Equivalente a la función "cero"
function int uno()
begin
i = 1;return i;
end

//Especificamos explícitamente que el valor a devolver es BYTE
function byte dos()
begin
b = 2;return b;
end

//Especificamos explícitamente que el valor a devolver es WORD
function word tres()
begin
w = 3;return w;
end

//Especificamos explícitamente que el valor a devolver es DWORD
function dword cuatro()
begin
dw = 4;return dw;
end

//No devuelve lo que uno esperaría. No funciona
function float cinco()
begin
f = 5.5; return f;
```

```
end

//No devuelve lo que uno esperaría. No funciona
function string seis()
begin
c="Seis"; return c;
end
```

Fíjate que en los dos últimos casos, cuando la función devuelve float ó string, parece no retornar bien el valor esperado. Esto es por un detalle que no habíamos comentado hasta ahora: las funciones que no devuelvan tipos de datos enteros (por entero se entiende la familia entera:byte, int, word, dword) se han de declarar previamente al código del proceso "Main()" mediante la sentencia DECLARE. Ya cuando hablamos de las variables públicas comentamos la existencia de esta sentencia, que servía para hacer saber a Bennu de la existencia de un proceso antes de llegar a su definición, escrita posteriormente en el código. Pues bien, podemos hacer lo mismo con funciones (y de hecho, para el caso de valores de retorno no enteros, es obligatorio hacerlo), y declarar las variables públicas y privadas de nuestra función previamente a la escritura de su código. Para utilizar la sentencia DECLARE con funciones, su sintaxis es la siguiente:

**DECLARE FUNCTION tipoDatoRetornado nombreFuncion (tipo param1,tipo param2,...)**
**...**
**END**

donde los parámetros que se especifiquen serán los mismos que tiene la propia función, y tipoDatoRetornado es el tipo de datos del valor (si es INT no es necesario ponerlo) que será devuelto con la sentencia RETURN (si es que ésta lo hace). Así pues, para que el ejemplo anterior funcionara correctamente con las funciones "cinco()" y "seis()", deberíamos añadir las siguientes líneas justo antes de la línea *process main()*.

```
Declare function float cinco()
End
Declare function string seis()
End
```

Otro ejemplo sobre el uso de DECLARE:

```
Import "mod_text";
Import "mod_key";
Import "mod_proc";

Declare Function string ejemplofunc( int param1)
    Private
        int privint;
    End
End

Process Main()
Private
          string texto;
End
Begin
      texto=ejemplofunc(1);
          write(0,150,10,4,texto);
          texto=ejemplofunc(2);
          write(0,150,20,4,texto);
          loop
             if(key(_esc)) exit(); end
             frame;
          end
End

Function string ejemplofunc( int param1)
Begin
    privint=param1;
    if (privint==1) return ("¿Qué taaaal?!!!");end
    if (privint==2) return ("¡¡¡Muy bien!!!!");end
```

```
End
```

En el ejemplo anterior hemos usado la sentencia DECLARE para declarar una variable privada (*privint*) de la función "ejemplofunc()".

Si lo que queremos es otra cosa totalmente diferente, que es utilizar dentro de una función una variable pública de un proceso externo, lo deberíamos hacer de la siguiente manera:

```
DECLARE PROCESS coche()
   PUBLIC
      Int gasolina;
   END
END

FUNCTION coche_llenar_gasolina(coche idcoche)
BEGIN
   idcoche.gasolina=100;
END
```

En este caso, tenemos una función "coche_llenar_gasolina()" con un parámetro que será el identificador de una instancia de un supuesto proceso "coche()". Fíjate (ya lo hemos visto otras veces) que dicho parámetro está definido no como INT sino como el nombre del proceso:"coche". La función lo que hace es modificar el valor de una variable pública llamada "gasolina", de esa instancia concreta del proceso "coche".

## *La sentencia #DEFINE

Esta sentencia permite definir -"crear"- macros ( más exactamente, asignar un texto a una macro). Y una macro es simplemente una palabra que, una vez definida, al aparecer más adelante en el código es sustituida por un texto determinado, especificado también en la sentencia #define.

Una macro es similar a una constante, pero tiene alguna diferencia importante. La más evidente es que, a diferencia de en las constantes, los valores de las macros pueden ser expresiones aritméticas/lógicas donde pueden intervenir diferentes operandos, los cuales pueden tener un valor diferente cada vez que la macro sea utilizada, ya que éstas -otra diferencia con las constantes- pueden admitir parámetros. En este sentido, una macro se parecería más a la definición de una función extremadamente simple.

Las sentencias #DEFINE es recomendable siempre escribiarlas al principio de nuestro código, justo después de las líneas "import" y antes de cualquier sección de declaración de variables. Su sintaxis es la siguiente:

**#DEFINE Macro(param1,param2,...) ExpresiónAsignadaALaMacro**

Fíjate que no escribimos el punto y coma al final (aunque si lo escribes tampoco pasa nada Ni tampoco especificamos el tipo de los parámetros ya que siempre serán INT.Los paréntesis que incluyen los parámetros de la macro son opcionales.

Un ejemplo:

```
Import "mod_text";
Import "mod_key";

#define CALCULO 2+3

Process Main()
Begin
  write(0,100,100,4,4*CALCULO);
          loop
          if(key(_esc)) break; end
             frame;
          end
```

```
End
```

En este caso hemos definido una macro con el nombre de “CALCULO”, sin parámetros. Cada vez que aparezca la palabra CALCULO en el código será sustituida por su valor, que es 2+3. Pero fíjate en un detalle. El resultado mostrado es 11, ya que, evidentemente, 11=4*2+3. Pero, ¿qué resultado obtenemos si ejecutamos este código?

```
Import “mod_text”;
Import “mod_key”;

#define CALCULO (2+3)

Process Main()
Begin
  write(0,100,100,4,4*CALCULO);
          loop
         if(key(_esc)) break; end
            frame;
          end
End
```

Obtenemos 20, ya que 20=4*(2+3). Como puedes ver, es muy importante el uso de los paréntesis en expresiones matemáticas definidas en macros, porque un despiste nos puede hacer que obtengamos valores que no esperábamos.

Otro ejemplo, con una macro que admite parámetros:

```
Import "mod_text";
Import "mod_key";

#define CALCULO(res,a,b) res=a+b

Process Main()
Private
          int mivar;
End
Begin
  CALCULO(mivar,2,3);
  write(0,100,100,4,4*mivar);
  loop
         if(key(_esc)) break; end
            frame;
  end
End
```

En este caso hemos utilizado una macro casi como si de una función se tratara. Puedes ver que el resultado del cálculo lo asignamos a una variable *mivar*. Fíjate también ahora que poner paréntesis o no ponerlos es lo mismo, ya que *res* (y por tanto, *mivar*), siempre va a valer 5.

Podemos complicar las macros con parámetros hasta un nivel de sofisticación que lleguen a tener una versatilidad prácticamente igual a la de una función estándar con la ventaja además de que las macros, por su naturaleza, se ejecutan más rápidamente que las funciones. Por ejemplo, a ver si sabes interpretar lo que obtienes de ejecutar el siguiente código:

```
Import "mod_text";
Import "mod_key";

#define operacion(a,b,c) if(b==0) b=c+1; else a=b+1; end

process main()
private
 int var1,var2=3;
end
begin
operacion(var1,var2,3);
write(0,100,100,4,var1);
```

```
write(0,100,120,4,var2);
repeat frame; until(key(_esc))
end
```

Por cierto, para que no cometas posibles errores en tus futuras macros, has de saber que si asignas algún valor a alguno de sus parámetros dentro de la macro (como hacemos en el ejemplo anterior con *a* y *b*), cuando invoques a la macro dentro del código, el lugar que ocupan dichos parámetros deberá ser rellenado con nombres de variables (que recogerán el valor asignado correspondiente) y no con valores explícitos, porque si no dará un error. De hecho, esto es una comportamiento lógico y esperable, ya que si lo piensas bien, no tiene sentido hacerlo de otra manera.

Seguramente te estarás preguntando por qué esta sentencia tiene ese misterioso símbolo "#" delante. Bien,este símbolo indica que el "comando" DEFINE en realidad es lo que se llama una directiva de preprocesador. El preprocesador es un pequeño programa que filtra el contenido de tu código antes de pasárselo al compilador, y una de sus misiones es interpretar ciertas directivas que en realidad pueden estar presente en cualquier punto del programa -aunque hemos recomendado por claridad que estén al principio de todo- y que se caracterizan por ocupar una línea completa y empezar por el carácter # al principio de la misma (opcionalmente, precedido por espacios). Así pues, cada vez que el preprocesador encuentre una línea que contiene una macro previamente definida, la sustituye por su valor real, y una vez ha acabado todo este proceso, es cuando transfiere el código fuente al compilador para que éste realice por fin su función.

También se pueden definir macros desde la línea de comandos, sin necesidad de incluir ninguna línea #DEFINE dentro del código fuente, gracias al parámetro adecuado del compilador *bgdc*. Consulta el capítulo 1 para recordar cómo se hacía.

Aparte de la directiva #DEFINE, existen unas cuantas más, como por ejemplo el bloque *#IFDEF variable/#ENDIF*, que sirve para crear secciones de código compilables opcionalmente sólo si *variable* está definida (es decir, zonas de código en el interior de este bloque que sólo se compilan si *variable* existe previamente). Dicha variable se puede haber definido anteriormente en el código, o haberse definido mediante el parámetro adecuado del compilador en el mismo momento de invocarlo por línea de comandos. De esta manera se pueden por ejemplo habilitar ó deshabilitar partes de código seleccionando funcionalidades diferentes para un proyecto (por ejemplo, en modo demo o modo release) dependiendo de una condición concreta. También existe el bloque *#IFNDEF variable/#ENDIF* , que comprueba justamente la condición contraria, que *variable* no esté definida, para compilar el código existente en su interior. En ambos bloques también se puede incluir una sección *#ELSE*. Es decir, podríamos tener un bloque como éste, por ejemplo:

```
#ifdef mivar
    import "mod_wm";
#else
    import "mod_scroll";
#endif
```

# CAPITULO 5

## Estructuras de datos

Las clases de datos que hemos explicado en el capítulo anterior (las variables de todo tipo y las constantes) son los únicos que hemos utilizado en los ejemplos hasta ahora. Éstos son los datos más simples: un nombre asociado con valor guardado en una determinada posición de la memoria del ordenador, cuyo contenido puede variar (variables) o no (constantes). No obstante, existen tipos más complejos de datos: las tablas y las estructuras.

### *Tablas

Una tabla es una lista de variables. Es decir, una tabla es un nombre que se relaciona, no con una sola posición de memoria (un solo valor) sino con tantas (tantos valores) como sea necesario.

Igual que las variables, las tablas pueden ser globales,locales, privadas o públicas.

Para definir en un programa, por ejemplo, una tabla global formada por números enteros, se puede hacer una declaración –e inicialización- como la siguiente:

```
GLOBAL
    Int mitabla1[3]=0,11,22,33;
END
```

Estas líneas declararían en un programa *mitabla1* como una lista de 4 variables enteras. Se debe tener en cuenta que se comienza a contar siempre desde la posición 0, y esta tabla tiene hasta la posición 3, como indica el número entre los símbolos [ ] (que se denominan corchetes y no se deben confundir con los paréntesis): ese número indica pues el número de la última posición. En este ejemplo, cuatro posiciones de la memoria del ordenador se reservarían para *mitabla1*, y se pondría inicialmente (antes de comenzar el programa) los valores, 0, 11, 22 y 33 en dichas posiciones.

Esta inicialización no es necesaria: también se podría haber declarado la tabla y ya está:

```
GLOBAL
    Int mitabla1[3];
END
```

en cuyo caso todos los valores de los elementos de la tabla valdrían 0 por defecto.

Hay que saber que si en la declaración de la tabla a la vez se inicializa (es decir, ya se introducen los valores de los elementos, como es el primer caso tratado), el número entre corchetes es opcional ponerlo, ya que si por ejemplo en ese momento se introdujeran cinco valores, Bennu reservaría sitio para cinco elementos en esa tabla, ni uno más ni uno menos, igual que si hubiéramos escrito un "4" entre los corchetes.

Evidentemente, nada evita que se pueda inicializar una tabla con un determinado número de valores y poner entre corchetes un número mayor de elementos: todos aquellos elementos que no sea rellenados por los valores en ese momento serán puestos a 0.

Cuando en un programa se tiene que consultar o modificar una de estas posiciones de memoria, se debe indicar el nombre de la tabla y, entre los corchetes, un valor numérico para especificar qué posición de la tabla se quiere consultar o modificar. Por ejemplo, una sentencia para poner el valor 999 en la posición 1 de *mitabla1* (que inicialmente valdría 11) sería como sigue:

```
mitabla1[1]=999;
```

Asimismo, sería posible definir tablas de mas dimensiones, mediantes corchetes extra; por ejemplo, en una tabla global entera la siguiente declaración...

```
GLOBAL
    Int mitabla2[2][4];
END
```

...crearía una tabla con 3 filas x 5 columnas = 15 posiciones de memoria.

Fíjate que todos los elementos de cualquier tabla han de ser del mismo tipo: o enteros, o decimales, o cadenas… No puede haber dentro de una misma tabla elementos de distintos tipos.

Las tablas son muy útiles cuando necesitamos almacenar muchos valores que de alguna manera están relacionados entre sí por su significado o propósito. Podríamos trabajar igualmente con variables "normales", pero es mucho más fácil declarar una única tabla (un nombre) con 100 posiciones por ejemplo que no declarar 100 variables diferentes (100 nombres) para guardar cada uno de esos 100 valores.

Un sinónimo de "tabla" es la palabra "array". También verás que a veces se les llama "vectores" a las tablas de una dimensión, y "matrices" a tablas de dos o más dimensiones

Pongamos un ejemplo donde se vea todo más claro. En el siguiente código (imaginemos que estamos dentro de una sección PRIVATE/END) hemos declarado cinco tablas, cada una de forma distinta.

```
int tabla1[]=5,3,65;
int tabla2[4];
string tabla3[]="hola","que","tal";
string tabla4[4];
int tabla5[2][3];
```

*La primera es una tabla (un vector) de elementos enteros a la que inicializamos en el mismo momento de declararla; al no haber especificado entre corchetes el número de elementos, ésta tendrá los mismos que valores se le introduzcan en la inicialización, es decir, 3.
*La segunda también es un vector de enteros, pero con un número determinado de elementos -cinco- y que no se inicializa, por lo que los valores iniciales de estos cinco elementos será 0.
*La declaración de la tercera tabla es similar a la de la primera con la salvedad de que aquélla es un vector de cadenas.
*De igual manera, la cuarta tabla es una tabla de cinco cadenas, inicialmente vacías.
*La quinta tabla es una tabla bidimensional (una matriz). Imagínatela como si fuera la rejilla de una hoja de cálculo o similar, formada por celdas dispuestas en filas y columnas. Cada celda vendrá identificada por dos índices, que vienen dados por los números dentro de los dos corchetes; el primer corchete indicará el número de la fila donde está situada la celda, y el segundo corchete indicará el número de la columna donde está situada la celda, empezando siempre por 0, y desde arriba hacia abajo y desde izquierda a derecha.. En la declaración de la tabla, el primer corchete indicará pues el último elemento de todas las filas -por lo tanto, nuestra tabla tiene tres filas- y el segundo corchete indicará el último elemento de todas las columnas -por lo tanto, nuestra tabla tiene cuatro columnas-.

Las dos preguntas más evidentes cuando trabajamos con tablas es: ¿cómo recupero los distintos valores de todos los elementos de una tabla? y ¿cómo puedo asignar nuevos valores a dichos elementos?

La respuesta a la primera pregunta es: mediante un bucle que recorra la tabla elemento a elemento y vaya trabajando como sea necesario con el valor que obtiene en cada iteración. El bucle que se utiliza más típicamente a la hora de recorrer tablas es el bucle FOR, porque sabemos cuándo empezamos (en el primer elemento) y sabemos cuándo acabamos (en el último elemento, que sabemos qué posición ocupa). Así, la manera más normal de recorrer un tabla -la *tabla1* anterior, en este caso- y mostrar por ejemplo por pantalla los distintos valores de los elementos por los que se va pasando, sería algo así:

```
for (i=0;i<3;i++)
      write(0,100,10+10*i,4,tabla1[i]);
end
```

Fíjate en un detalle muy importante. Este FOR va a realizar tres iteraciones, cuando *i* valga 0, 1 y 2, ya

que en el segundo parámetro del for se especifica que *i* ha de ser siempre MENOR que 3. Y lo hemos hecho así precisamente porque sabemos que *tabla1* tiene sólo 3 elementos, tabla1[0], tabla1[1] y tabla[2]. Es decir, que hubiera sido un error haber puesto algo así como (es bastante frecuente que pase):

```
for (i=0;i<=3;i++)
      write(0,100,10+10*i,4,tabla1[i]);
end
```

ya que en este bucle se haría una iteración de más, en la cual tabla1[3] no existe y provocaría un error.Hay que tener ojo con esto.

De igual manera, podemos recorrer cualquier otro tipo de tablas, como las de cadenas. Si probamos con *tabla4*:

```
for (i=0;i<5;i++)
      write(0,100,10+10*i,4,tabla4[i]);
end
```

Veremos, no obstante, que no aparece nada en pantalla. Esto es porque esta tabla está vacía y no tiene ningún valor que mostrar (a diferencia de las tablas numéricas, las tablas de cadena no se rellenan automáticamente con ningún valor).

Podemos ya por tanto escribir un programa completo que recorra las cuatro tablas unidimensionales de este ejemplo y muestre por pantalla los valores almacenados en ellas (si es que tienen), así:

```
Import "mod_text";

process main()
private
           int tabla1[]=5,3,65;
           int tabla2[4];
           string tabla3[]="hola","que","tal";
           string tabla4[4];
           int tabla5[2][3];
           int i;
end
begin
//Recorremos la primera tabla
  for (i=0;i<3;i++)
           write(0,10,10+10*i,4,tabla1[i]);
  end

//Recorremos la segunda tabla (rellena de ceros)
  for (i=0;i<5;i++)
           write(0,50,10+10*i,4,tabla2[i]);
  end

//Recorremos la tercer tabla
  for (i=0;i<3;i++)
           write(0,90,10+10*i,4,tabla3[i]);
  end

//Recorremos la cuarta tabla (vacía)
    for (i=0;i<5;i++)
           write(0,130,10+10*i,4,tabla4[i]);
   end
loop
           frame;
end
end
```

La segunda pregunta que teníamos era que cómo podemos asignar nuevos valores a los elementos de una tabla. Es bien sencillo. Tal como se ha comentado anteriormente, si se quiere cambiar el valor de un elemento concreto, simplemente hay que asignarle su valor como haríamos con cualquier otra variable. Por ejemplo, así:

```
tabla1[1]=444;
```

asignaríamos el valor 444 al elemento 1 (el segundo) de la tabla *tabla1*. Si quisiéramos cambiar los valores de un conjunto de elementos de la tabla, otra vez tendríamos que recurrir a un bucle para ir elemento a elemento y cambiar su valor. Por ejemplo, si haces así:

```
for (i=0;i<3;i++)
          tabla1[i]=444;
          write(0,100,10+10*i,4,tabla1[i]);
end
```

verás como antes de imprimirse por pantalla los valores de los elementos de *tabla1* se les da a todos por igual el valor de 444, sobreescribiendo el valor anterior que tuvieran. Si introduces este bucle en el ejemplo anterior podrás comprobar como se imprimirán 3 números 444 correspondientes a los 3 elementos de *tabla1*.

Pero, ¿qué pasa con la tabla bidimensional? ¿Cómo se asignan los valores? Podríamos asignarlos a la vez que se declara:

```
Import "mod_text";

process main()
private
  int tabla5[2][3]=1,2,3,4,5,6,7,8;
end
begin
  write(0,100,10,4,tabla5[0][0]);
  write(0,100,20,4,tabla5[0][1]);
  write(0,100,30,4,tabla5[0][2]);
  write(0,100,40,4,tabla5[0][3]);
  write(0,100,50,4,tabla5[1][0]);
  write(0,100,60,4,tabla5[1][1]);
  write(0,100,70,4,tabla5[1][2]);
  write(0,100,80,4,tabla5[1][3]);
  write(0,100,90,4,tabla5[2][0]);
  write(0,100,100,4,tabla5[2][1]);
  loop
           frame;
  end
end
```

Fíjate el orden que se sigue para ir llenando los elementos: primero es el [0][0], después el [0][1], después el [0][2], luego el [1][0], y así.

También se puede asignar un valor en mitad del código a un elemento concreto:

```
tabla5[1][2]=444;
```

O también se puede asignar nuevos valores a múltiples elementos recorriendo de arriba a abajo todos los elementos de la matriz. Y ¿eso cómo se hace? Si piensas un poco, lo deducirás. Si hemos necesitado un bucle para recorrer los elementos uno detrás de otro de un vector, necesitaremos dos bucles para recorrer las filas por un lado y las columnas por otro de esa matriz bidimensional. No obstante, la forma correcta de hacerlo puede que te sorprenda un poco: estos dos bucles irán uno dentro de otro (lo que se llaman bucles anidados). Vamos a poner primero un ejemplo de recorrido de una matriz para leer los valores de sus elementos (hace lo mismo que el ejemplo anterior pero usando bucles), y luego pondremos otro ejemplo que no los lea, sino que los modifique por un valor dado. El ejemplo de lectura es:

```
Import "mod_text";

process main()
private
    int tabla5[2][3]=1,2,3,4,5,6,7,8;
    int i,j,a=10;
end
begin
```

```
  for(i=0;i<3;i++)
        for(j=0;j<4;j++)
             write(0,100,10+a ,4,tabla5[i][j]);
             a=a+10;
        end
  end
  loop
        frame;
  end
end
```

El ejemplo de escritura es simplemente añadir una línea al anterior:

```
Import "mod_text";

process main()
private
    int tabla5[2][3]=1,2,3,4,5,6,7,8;
    int i,j,a=10;
end
begin
  for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            tabla5[i][j]=444;
             write(0,100,10+a ,4,tabla5[i][j]);
             a=a+10;
        end
  end
  loop
        frame;
  end
end
```

Y los dos bucles ¿por qué? En la primera iteración del bucle más exterior, i=0. En la primera iteración del bucle más interior, j=0. A partir de aquí, se van a ir sucediendo iteraciones en el bucle interior, de manera que mientras *i* vale 0, j valdrá consecutivamente 1,2 y 3. Es decir, se abrán realizado cuatro iteraciones con los valores i=0 y j=0, i=0 y j=1, i=0 y j=2 y i=0 y j=3. En este momento, se ha llegado al límite de iteraciones en el bucle interior, se sale de él y se va a la siguiente iteración del bucle exterior, con lo que i=1, y se vuelve a repetir el proceso de las cuatro iteraciones internas con i=1, por lo que ahora tendremos los valores i=1 y j=0, i=1 y j=1, i=1 y j=1 y i=1 y j=3. Y así hasta llegar a la última iteración del bucle exterior, que es cuando i=2. Si te fijas, con este método hemos logrado recorrer todos los elementos de la matriz.

Y con una matriz tridimensional ¿qué? Pues se usarán de forma similar tres bucles anidados.

Un ejemplo práctico de uso de tablas (para implementar por ejemplo un sistema de diálogos entre personajes de un juego):

```
Import "mod_text";
Import "mod_key";
Import "mod_proc";

process main()
Private
  int contador=0;
// Todos los textos que aparecerán
  string textosjuego[]=
    "Volver...",
    "con la frente marchita",
    "las nieves del tiempo",
    "platearon mi sién.",
    "Sentir...",
    "que es un soplo la vida,",
    "que veinte años no es nada,",
    "¡qué febril la mirada!",
    "Errante, la sombra",
    "te busca y te nombra.",
```

```
    "Vivir...",
    "con el alma aferrada",
    "a un dulce recuerdo",
    "que lloro otra vez.";
end
Begin
   Loop
      delete_text(0);
      If(key(_down))
           contador++;
           //Hasta que no se suelte la tecla no pasa nada
           while(key(_down) <> 0) frame; end
      End
      //Muestro la cadena que ocupa la posición dada por el índice "contador"
      write(0,0,0,0,textosjuego[contador]);
      //Si llego al final de la tabla, vuelvo a empezar
      If(contador=>14) contador=0;End
      If(key(_esc)) exit();End
      Frame;
  End
End
```

## *Estructuras y tablas de estructuras

Una estructura es similar a una ficha típica de oficina que contiene información sobre una persona concreta como el número de identidad, el nombre, dirección, teléfono,etc. Normalmente no se utiliza una estructura sola, sino que se trabajan con tablas de estructuras: las tablas de estructuras serían entonces como el cajón que alberga todas las fichas/estructuras. A las estructuras también se les llama a menudo registros. Y cada anotación dentro de una ficha (el nombre, dirección, teléfono...) se denomina campo. Aunque no es formalmente correcto, es muy común denominar, para abreviar, estructura a lo que realmente es una tabla de estructuras.

Igual que las variables, las estructuras pueden ser globales,locales, privadas o públicas.

Por ejemplo, en un juego se podría definir la siguiente estructura global para guardar la información sobre la posición en pantalla de tres enemigos (en este caso, se declara e inicializa la estructura a la vez):

```
GLOBAL
           STRUCT posicion_enemigos[2]
                      Int coordenada_x;
                      Int coordenada_y;
           END = 10,50,0,0,90,80;
END
```

Este cajón de fichas se llamaría *posicion_enemigos*, contendría 3 fichas para 3 enemigos (con los números 0,1 y 2, como indica el 2 entre corchetes), y cada ficha dos anotaciones, la coordenada horizontal de un enemigo y la vertical. En la jerga correcta, *posicion_enemigos* es una estructura de 3 registros, cada uno con 2 campos, que serían *coordenada_x* y *coordenada_y*. La lista de valores siguiente posicionaría el primer enemigo en la coordenadas (10,50), el segundo en (0,0) y el tercero en (90,80).Como verás, el 10 es el valor de *coordenada_x* para posicion_enemigos[0], el 50 es el valor de *coordenada_y* para posicion_enemigos[0], el 0 es el valor de *coordenada_x* para posicion_enemigos[1],etc.

Fíjate como en este ejemplo,a la vez de declarar la estructura e inicializar con valores sus campos, al final de dichos valores se escribe un punto y coma. Si se declara la estructura y ya está (dejando inicialmente los campos vacíos), no se escribirá ningún punto y coma detrás del END, como es habitual.

Para acceder después en el programa a uno de estos valores numéricos se debería indicar el nombre de la estructura, el número de registro y el nombre del campo. Por ejemplo, para poner el valor 66 en la posición de memoria que guarda las coordenada vertical (y) del segundo enemigo (el registro número 1, ya que el primer enemigo tiene sus coordenadas en el registro 0), se utilizaría la siguiente sentencia:

```
posicion_enemigos[1].coordenada_y = 66;
```

Y para visualizarlo por pantalla, se podría utilizar por ejemplo:

```
write(0,100,100,4,posicion_enemigos[1].coordenada_y );
```

En realidad es muy parecido a las tablas, pero indicando después del número de registro el símpolo "." (un punto) y el nombre del campo.

La gran diferencia que tienen con las tablas es que los campos pueden no ser del mismo tipo.De hecho, cada campo de la estructura puede ser una variable, una tabla u otra estructura completa, con sus diferentes registros o campos. Por ejemplo, sería completamente correcto crear una estructura global como ésta:

```
GLOBAL
          STRUCT datos_clientes [9]
                    Byte codigo_interno;
                    String DNI;
                    String nombre;
                    String apellidos;
                    Int num_articulos_comprados;
                    Float dinero_que_debe;
          END
END
```

En este caso, habríamos creado 10 fichas para los datos de nuestros clientes, en las cuales incluiríamos, para cada uno de ellos, datos de diferente tipo. Pero ahora no hemos inicializado la estructura, sólo la hemos declarado. Se pueden rellenar los valores de los campos en cualquier momento. Si no se establecen en el momento de la declaración de la estructura, todos los valores de sus campos se inicializan a 0 por defecto.

Si quisiéramos darle valores a un registro entero concreto (por ejemplo el 2), ya sabemos que no tendríamos más que hacer:

```
datos_clientes[2].codigo_interno=34;
datos_clientes[2].DNI="89413257F";
datos_clientes[2].nombre="Federico";
datos_clientes[2].apellidos="Martínez Fernández";
datos_clientes[2].num_articulos_comprados=2;
datos_clientes[2].dinero_que_debe=789.04;
```

Veamos un ejemplo de código que utilice una tabla de estructuras para almacenar las posiciones de un determinado número de enemigos al principio de la partida y al final también. Asignaremos valores a los campos de cada registro y luego los comprobaremos mostrándolos por pantalla.La tabla de estructuras del ejemplo contendrá la posición inicial y final de hasta un máximo de 10 procesos enemigos (sería como una caja con 10 fichas, cada una de ellas indicando la x e y inicial y la x e y final de un proceso, es decir, cuatro campos).El ejemplo primero se encargará, mediante bucles que recorrerán las 10 estructuras, de asignar valores a todos los cuatro campos de cada registros. Y posteriormente los visualizará, utilizando para recorrer las 10 estructuras otra vez bucles.

```
Import "mod_text";
Import "mod_key";
Import "mod_video";

process main()
private
          struct movimiento_enemigos[9]
                    Int x_inicial;
                    Int y_inicial;
                    Int x_final;
                    Int y_final;
          end
          int i,a;
end
begin
  set_mode(750,200);
  for(i=0;i<10;i++)
```

```
          movimiento_enemigos[i].x_inicial=i;
          movimiento_enemigos[i].y_inicial=i;
          movimiento_enemigos[i].x_final=i+1;
          movimiento_enemigos[i].y_final=i+1;
  end

  for(i=0;i<10;i++)
    write(0,50+a,10,4,"Enemigo " + i);
          write(0,50+a,20,4,movimiento_enemigos[i].x_inicial);
          write(0,50+a,30,4,movimiento_enemigos[i].y_inicial);
          write(0,50+a,40,4,movimiento_enemigos[i].x_final);
          write(0,50+a,50,4,movimiento_enemigos[i].y_final);
          a=a+67;
  end

repeat
          frame;
until (key(_esc))
end
```

Otro ejemplo más complicado. Si se hace la siguiente declaración:

```
STRUCT a[2]
     int b;
     int c[1];
END=1,2,3,4,5,6,7,8,9;
```

Los valores inicializados, ¿a qué campo van a parar? Sabemos que la estructura *a[]* tiene 3 registros (desde a[0] hasta a[2]) y en cada registro 3 campos (b, c[0] y c[1]), luego la anterior declaración inicilizaría la estructura de la siguiente forma:

a[0].b = 1;
a[0].c[0] = 2;
a[0].c[1] = 3;
a[1].b = 4;
a[1].c[0] = 5;
a[1].c[1] = 6;
a[2].b = 7;
a[2].c[0] = 8;
a[2].c[1] = 9;

Se puede comprobar haciendo el bucle pertinente que recorra los elementos y los muestre por pantalla, como ya sabemos.

## *La orden "SizeOf()"

Esta función devuelve el número de bytes que ocupa en memoria el tipo de dato, tabla o estructura (o tabla de estructuras) que se le pase por parámetro. Se introduce esta función en este capítulo porque es muy útil y práctica en muchos casos donde se necesita saber el tamaño total que ocupa determinada tabla o estructura (caso bastante frecuente). Por ejemplo, sabemos que un dato Int ocupa 4 bytes en memoria, pues la sentencia

```
variable=sizeof(int);
```

debería de asignar el valor de 4 a "variable". Y así con todos los tipos de datos, incluyendo además tablas y estructuras. Si tenemos una tabla de 20 elementos Int, ¿cuánto ocupará en memoria? Pues 20x4=80 bytes, por lo que la sentencia

```
variable=sizeof(mitabla);
```

debería de asignar el valor de 80 a "variable".

Hay que hacer notar que **sizeof()** devuelve el tamaño TOTAL de la tabla en memoria, el cual se calcula a partir de sumar el tamaño del total de elementos, ya sean elementos que hayamos llenado nosotros explícitamente como también aquellos que aparentemente estén vacíos, pero que en realidad no lo están porque ya sabemos que si no se les asigna un valor explícitamente, los elementos de una tabla siempre son asignados al valor 0 (o en el caso de datos de tipo cadena, una cadena nula), por lo que siempre tendremos las tablas llenas completamente y su tamaño en memoria siempre será el mismo tenga más o menos elementos rellenados "a mano".

Igualmente pasa con estructuras. Si tenemos una estructura llamada *miestructura*, simplemente deberíamos de escribir *sizeof(miestructura)* para saber cuánto ocupa en memoria esa estructura.

Ojo, porque si estamos hablando de una tabla (ya sea de elementos simples como de estructuras con varios registros) llamada por ejemplo *mitabla*, *sizeof(mitabla)* devolverá el tamaño completo que ocupan todos los registros que forman parte de la tabla, y en cambio *sizeof(mitabla[0])* devolverá el tamaño que ocupa sólo el primer registro -que es el mismo que cualquiera de los otros-. A partir de aquí es fácil ver que una operación -bastante común- como esta división:

```
sizeof(mitabla)/sizeof(mitabla[0])
```

lo que devolverá será el número de elementos que puede albergar dicha tabla.

Veamos un ejemplo clarificador:

```
Import "mod_text";
Import "mod_key";
Import "mod_video";

process main()
private
   byte a;
   short b;
   word c;
   int d;
   dword e;
   float f;
   string g;
   char h;
   int tabla1[3];
   byte tabla2[1][1];
   struct struct1
          int a;
          string b;
   end
   struct struct2[9]
          int a;
          float b;
          string c;
   end
end
begin
set_mode(640,480);
write(0,300,10,4,"Byte: " + sizeof(byte));
write(0,300,20,4,"Short:" + sizeof(short));
write(0,300,30,4,"Word: " + sizeof(word));
write(0,300,40,4,"Int: " + sizeof(int));
write(0,300,50,4,"Dword: " + sizeof(dword));
write(0,300,60,4,"Float: " + sizeof(float));
write(0,300,70,4,"String: " + sizeof(string));
write(0,300,80,4,"Char: " + sizeof(char));
write(0,300,90,4,"Vector de 4 enteros: " + sizeof(tabla1));
write(0,300,100,4,"Matriz de bytes 2x2: " + sizeof(tabla2));
write(0,300,110,4,"Estructura con un campo Int y otro String: " + sizeof(struct1));
write(0,300,120,4,"Tabla de 10 estructuras con un campo Int, otro Float y otro String: "
+ sizeof(struct2));
write(0,300,130,4,"Un registro de la tabla de 10 estructuras anterior: " +
sizeof(struct2[0]));
```

```
loop
    if(key(_esc)) break; end
    frame;
end
end
```

Y otro ejemplo interesante:

```
Import "mod_text";
Import "mod_key";
Import "mod_video";

process main()
private
 int Array1[9]; //Tiene 10 elementos reservados en memoria, todos llenos de 0
 float Array2[]= 1.0,2.0,3.0; /*Tiene 3 elementos reservados en memoria, ya que al no
especificar un índice entre los corchetes, se reservarán tantos elementos como valores se
especifiquen*/
 int SizeArray;
end
begin
 set_mode(320,240);
 SizeArray= sizeof(Array1)/sizeof(Array1[0]) ;
 write(0,150,50,4,"El tamaño de Array1 es: " + SizeArray); //Ha de mostrar un 10
 SizeArray= sizeof(Array2)/sizeof(Array2[0]) ;
 write(0,150,150,4,"El tamaño de Array2 es: " + SizeArray); //Ha de mostrar un 3
 repeat frame; until(key(_esc))
end
```

La función **sizeof()** también se puede usar para averiguar el espacio que ocupan lon datos definidos por el usuario con TYPE (visto a continuación).

## *Los tipos definidos por el usuario. Cláusula TYPE

Un paso más allá en el concepto de estructuras es el de tipo definido por el usuario (TDU). Un TDU es una estructura/ficha/registro (ojo, no una tabla de estructuras), con una diferencia: no es ninguna variable y por tanto no se puede usar directamente, sino que se utiliza para definir a su vez a otras variables. Es decir, el tipo de estas otras variables no será ni Int ni float ni ningún otro tipo predefinido, sino que será precisamente el TDU, el cual es un tipo de datos formado a partir de otros. Y estas variables podrán ser incluso tablas de elementos de ese TDU.

Es decir, resumiendo, el bloque TYPE/END declara un tipo de dato definido por el usuario y compuesto a su vez por varios datos de otro tipo, y no crea ninguna variable concreta en el momento en el que aparece: simplemente, le otorga un significado al identificador utilizado y permite que sea usado posteriormente en cualquier declaración de otras variables

Para crear un TDU se utiliza el bloque TYPE/END. Su sintaxis es exactamente igual a la de un bloque STRUCT/END pero sin inicialización de valores, y con la salvedad que se ha de escribir obligatoriamente fuera de cualquier sección de código de proceso y antes de las posibles declaraciones GLOBAL/END ó LOCAL/END de variables (aunque después de las líneas "import" y los posibles #DEFINE) .

Un uso típico de las tablas de TDU es por ejemplo, para almacenar en cada registro determinada información sobre un enemigo (gráfico,posición x e y, vida, nivel...) , o una pieza de un rompecabezas (gráfico, posición x e y, ancho, alto...), etc. Ya verás que sus aplicaciones son múltiples, y se te irán ocurriendo más cuando los necesites.

Veamos un ejemplo. En el siguiente código se crea un TDU compuesto de un campo entero, otro decimal y otro de cadena. También se crea otro TDU compuesto de una tabla de enteros y de una tabla de caracteres. A partir de aquí, en la sección PRIVATE/END se declaran dos variables del primer TDU y una del segundo. El programa lo único que hace es asignar valores a los campos de las tres variables declaradas y seguidamente mostrarlos por pantalla.

```
Import "mod_text";
Import "mod_key";

type tipo1
          int a;
          float b;
          string c;
end
type tipo2
          int a[20];
          char b[15];
end


process main()
private
   tipo1 hola,adios;
   tipo2 quetal;
end
begin

hola.a=2;
hola.b=4.5;
hola.c="Lalala";

adios.a=87;
adios.b=96.237;
adios.c="Lerele";

quetal.a[0]=3;
quetal.a[1]=64;
quetal.a[2]=9852;
quetal.b[0]="c";
quetal.b[1]="g";

write(0,100,10,4,hola.a);
write(0,100,20,4,hola.b);
write(0,100,30,4,hola.c);
write(0,100,40,4,adios.a);
write(0,100,50,4,adios.b);
write(0,100,60,4,adios.c);
write(0,100,70,4,quetal.a[0]);
write(0,100,80,4,quetal.a[1]);
write(0,100,90,4,quetal.a[2]);
write(0,100,100,4,quetal.b[0]);
write(0,100,110,4,quetal.b[1]);

repeat
          frame;
until(key(_esc))
end
```

Como se puede ver, la mayor utilidad de los TDU es poder crear un tipo estructura con un determinado nombre identificador para declarar posteriormente estructuras a partir simplemente de ese identificador, y cuyo contenido será el escrito en el interior del bloque TYPE. Se puede deducir de esto que TYPE es mucho más cómodo que STRUCT si se tienen que crear múltiples variables en lugares diferentes del programa con la misma estructura.

En definitiva,si, por ejemplo, al principio del programa aparece este código:

```
TYPE color
   byte r,g,b;
END
```

sería posible declarar una o más variables del nuevo tipo "color" en cualquier sección de datos, por ejemplo GLOBAL:

```
GLOBAL
   color p;
```

```
END
```

siendo en este ejemplo, la variable *p* es una estructura con tres componentes de tipo byte (r, g, b).

## *Copia y comparación entre dos tablas o dos estructuras. Gestión básica de memoria

En ciertas ocasiones nos puede interesar copiar el contenido completo de una tabla en otra, o de una estructura a otra. Lo más inmediato que se nos ocurriría sería probar esto:

```
Import "mod_text";
Import "mod_key";

process main()
private
struct struct1
          Int c;
          Int d;
end =1,4;
struct struct2
          Int c;
          Int d;
end
int tabla1[1]=6,2;
int tabla2[1];
end
begin
//Las estructuras han de ser idénticas en lo que se refiere a nº,tipo y orden de campos
  struct2=struct1;   //Copio los valores de los campos de struct1 a struc2
/*Los elementos de las dos tablas han de ser del mismo tipo, y el tamaño de tabla2 ha de
ser igual o mayor al de tabla1.*/
  tabla2=tabla1;    //Copio los valores de los elementos de tabla1 a tabla2

  //Muestro que efectivamente se han copiado bien.
  write(0,100,10,4,struct2.c); //Debería de salir un 1
  write(0,100,20,4,struct2.d); //Debería de salir un 4
  write(0,100,30,4,tabla2[0]); //Debería de salir un 6
  write(0,100,40,4,tabla2[1]); //Debería de salir un 2

repeat
   frame;
until (key(_esc))
end
```

Bien. Si intentamos ejecutar esto el intérprete nos devolverá un error en la línea donde se supone que copio el contenido de struct1 a struct2. Parece pues que no se hace así la copia. Si comentamos esa línea para probar al menos la copia de las tablas, veremos otra cosa curiosa: el programa se ejecuta sin errores pero resulta que sólo se ha copiado el valor del primer elemento: los valores del resto de elementos no. O sea que tampoco funciona copiar tablas de esta manera.

Copiar tablas o estructuras no es tan fácil como copiar valores entre variables simples.Para lograr hacerlo correctamente, necesitamos hacer uso de la función **memcopy()**, definida en el módulo "mod_mem".

**MEMCOPY(&DESTINO,&ORIGEN,BYTES)**

Esta función, definida en el módulo "mod_mem", copia un bloque de memoria de tamaño BYTES desde la tabla/estructura especificada por su nombre ORIGEN (precedido del símbolo &), a la tabla/estructura especificada por su nombre DESTINO (precedido del símbolo &).

*Te estarás preguntando por qué hay que poner el & delante de los dos primeros parámetros de este comando (y de muchos otros que iremos viendo). Este símbolo indica que no se trabaja con valores sino con sus direcciones de memoria.*

*La memoria del ordenador está organizada en forma de casillas, donde cada una tiene una dirección,gracias a la cual el ordenador sabe dónde tiene guardados los datos: la memoria RAM del ordenador es como un inmenso apartado de correos. Cada vez que veas un símbolo & delante del nombre de alguna variable, estaremos utilizando no el valor que contiene esa variable, sino la dirección de memoria donde está alojado esa variable (y por tanto, su valor). De hecho, por eso el comando* ***memcopy()*** *se llama así: porque lo que hace en realidad es copiar un bloque entero de memoria de un tamaño dado por BYTES, desde una dirección de memoria dada por donde empieza a guardarse ORIGEN, a una dirección de memoria que es donde empieza a guardarse DESTINO, sobrescribiéndolo en este caso. El tipo de datos POINTER ("puntero") es el requerido para este tipo de parámetros*

*En la práctica lo único que hay que acordarse cuando se quiera utilizar esta función es añadir un símbolo & delante de los nombres -ojo, nada de valores concretos- de las variables estándar que serán el primer y segundo parámetro de la función*

Ambas zonas de memoria ORIGEN y DESTINO pueden solaparse, en cuyo caso la zona de destino contendrá tras la ejecución los valores que antes estaban en la de origen, machacando cualquier posición compartida.

PARAMETROS:

POINTER DESTINO : Puntero a la zona de destino
POINTER ORIGEN: Puntero a la zona de origen
INT BYTES : Número de bytes a copiar

Es decir, que para copiar el contenido de una tabla a otra (o de una estructura a otra), lo que se tendrá que hacer es copiar el bloque de memoria donde está almacenada la tabla/estructura de origen a otro sitio de la memoria del ordenador, el cual será el bloque de destino. Parece muy extraño, pero en verdad es muy sencillo si volvemos a intentar el ejemplo anterior haciendo uso de esta nueva función:

```
Import "mod_text";
Import "mod_key";
Import "mod_mem";

process main()
private
struct struct1
          Int c;
          Int d;
end =1,4;
struct struct2
          Int c;
          Int d;
end
int tabla1[1]=6,2;
int tabla2[1];

end
begin

  //Las estructuras han de ser idénticas en lo que se refiere a nº,tipo y orden de campos
  //Copio los valores de los campos de struct1 a struct2
  memcopy(&struct2,&struct1,sizeof(struct1));

  /*Los elementos de las dos tablas han de ser del mismo tipo, y el tamaño de tabla2 ha
de ser igual o mayor al de tabla1.*/
  //Copio los valores de los elementos de tabla1 a tabla2
  memcopy(&tabla2,&tabla1,sizeof(tabla1));

  //Muestro que efectivamente se han copiado bien.
  write(0,100,10,4,struct2.c); //Debería de salir un 1
  write(0,100,20,4,struct2.d); //Debería de salir un 4
  write(0,100,30,4,tabla2[0]); //Debería de salir un 6
  write(0,100,40,4,tabla2[1]); //Debería de salir un 2

  //Muestro los valores de la tabla y estructura originales, que quedan como estaban
```

```
  write(0,100,60,4,struct1.c); //Debería de seguir saliendo un 1
  write(0,100,70,4,struct1.d); //Debería de seguir saliendo un 4
  write(0,100,80,4,tabla1[0]); //Debería de seguir saliendo un 6
  write(0,100,90,4,tabla1[1]); //Debería de seguir saliendo un 2

repeat
  frame;
until (key(_esc))
end
```

Podemos comprobar que ahora sí se realiza la copia correctamente.

No es necesario utilizar la función **memcopy()** si solamente se desea copiar un campo concreto de una estructura a otra, o un elemento concreto de una tabla a otra. En estos casos, una simple asignación del tipo:

```
tabla2[indiceelementoconcreto]=tabla1[indiceelementoconcreto];
```

ó

```
struct2.uncampoconcreto=struct1.uncampoconcreto;
```

funcionará perfectamente. El uso de **memcopy()** está diseñado para copiar tablas o estructuras enteras (y también variables de TDU).

Una función similar a **memcopy()** es **memmove()**. Ambas tienen los mismos parámetros con el mismo significado y ambas copian el contenido de una tabla/estructura en otra. La diferencia entre ellas está en la manera de realizar esa copia: mientras **memcopy()** copia directamente el contenido de una tabla/estructura en otra, **memmove()** copia el contenido de la tabla/estructura origen en un buffer intermedio, para posteriormente copiarlo de ahí a la tabla/estructura destino. Esta diferencia de funcionamiento interno es relevante en el caso de que al copiar el contenido del área de memoria origen en el área de destino exista solapamiento; en este caso, el comportamiento de **memcopy()** no estaría definido pero sí el de **memmove()**.

Por otra parte, nos encontraremos con un problema similar al acabado de comentar en los párrafos precedentes si lo que deseamos es comparar el contenido completo de una tabla con otra, o de una estructura con otra. Lo más inmediato que se nos podría ocurrir podría ser esto:

```
Import "mod_text";
Import "mod_key";

process main()
private
struct struct1
          Int c;
          Int d;
end =1,4;
struct struct2
          Int c;
          Int d;
end = 3,8;
int tabla1[1]=6,2;
int tabla2[1]=6,5;
end
begin
  if (struct1 == struct2)
          write(0,100,100,4,"Todos los valores de las estructuras son iguales");
  else
          write(0,100,150,4,"Algún valor de las estructuras es diferente");
  end

  if (tabla1 == tabla2)
          write(0,100,200,4,"Todos los valores de las tablas son iguales");
  else
          write(0,100,250,4,"Algún valor de las tablas es diferente");
  end
repeat
```

```
          frame;
until(key(_esc))
end
```

Pero nos vuelve a pasar lo mismo. El if que compara las estructuras devuelve un error de compilación que no nos permite ejecutar el programa, y la comparación de tablas sólo compara el primer elemento, obviando los siguientes. No es así pues como se comparan tablas, estructuras o TDU. Necesitamos para ello la función **memcmp()**, definida en el módulo "mod_mem".

| **MEMCMP(&BLOQUE1,&BLOQUE2,BYTES)** |
|---|
| Esta función, definida en el módulo "mod_mem", compara el contenido de un bloque de memoria dado por el primer parámetro con el contenido de otro bloque de memoria dado por el segundo parámetro. Ambos bloques tendrán un tamaño idéntico en bytes, dado por el tercer parámetro.<br><br>Esta función devolverá 0 si los bloques son iguales, y -1 si no lo son<br><br>PARAMETROS:<br>POINTER BLOQUE1 : Puntero a comienzo de un bloque de memoria<br>POINTER BLOQUE2: Puntero al comienzo del otro bloque de memoria cuyo contenido se compará con el primero<br>INT BYTES : Tamaño en bytes de los bloques de memoria (es el mismo para los dos, lógicamente)<br><br>VALOR RETORNADO: INT : Resultado de la comparación (0 si son iguales, -1 si son diferentes) |

Sabiendo esto, modificaremos nuestro ejemplo para que ahora funcione:

```
Import "mod_text";
Import "mod_key";
Import "mod_mem";

process main()
private
struct struct1
          Int c;
          Int d;
end =1,4;
struct struct2
          Int c;
          Int d;
end = 3,8;
int tabla1[1]=6,2;
int tabla2[1]=6,5;
end
begin
  if (memcmp(&struct1,&struct2, sizeof(struct1))==0)
          write(0,100,100,4,"Todos los valores de las estructuras son iguales");
  else
          write(0,100,150,4,"Algún valor de las estructuras es diferente");
  end

  if (memcmp(&tabla1,&tabla2, sizeof(tabla1))==0)
          write(0,100,200,4,"Todos los valores de las tablas son iguales");
  else
          write(0,100,250,4,"Algún valor de las tablas es diferente");
  end
repeat
          frame;
until(key(_esc))
end
```

Otro ejemplo muy similar de la función **memcmp()**, pero esta vez utilizando tipos definidos por el usuario podría ser éste:

```
Import "mod_text";
Import "mod_key";
```

```
Import "mod_mem";

type tipoper
  int var1;
  int var2;
end

global
   tipoper v1;
   tipoper v2;
   int i;
end

process main()
begin
v1.var1 = 1;
v1.var2 = 2;
v2.var1 = 1;
v2.var2 = 3;
i=memcmp(&v1,&v2,sizeof(tipoper));
write(0,10,10,0,i);

v2.var1 = 1;
v2.var2 = 2;
i=memcmp(&v1,&v2,sizeof(tipoper));
write(0,10,20,0,i);

while(!key(_ESC))
   frame;
end
end
```

No es necesario utilizar la función **memcmp()** si solamente se desea comparar el valor contenido en un campo concreto de una estructura y en otra, o en un elemento concreto de una tabla y en otra. En estos casos, una simple comparación del tipo:

```
if(tabla2[indiceelementoconcreto]==tabla1[indiceelementoconcreto])
```

ó

```
if(struct2.uncampoconcreto==struct1.uncampoconcreto)
```

funcionará perfectamente. El uso de **memcmp()** está diseñado para comparar tablas o estructuras enteras (y también variables de TDU, como hemos visto).

## *Pasar un vector (tabla unidimensional) o un TDU como parámetro de un proceso/función

En páginas anteriores se ha explicado cómo pasar parámetros en un proceso o función. Pero estos parámetros eran siempre de tipos simple (enteros, decimales, cadenas). ¿Cómo se puede pasar un vector o un TDU? Es algo que nos interesará saber porque no se hace de la misma manera debido a que se utilizan direcciones de memoria y punteros, conceptos éstos bastante avanzados. De hecho, para entender completamente el proceso de paso por parámetro de vectores necesitaríamos tener más conocimientos sobre programación. No obstante, a pesar de no ser muy pedagógico, a continuación vamos a resumir en forma de receta los pasos que hay que hacer siempre para lograr conseguir nuestro objetivo, sin profundizar más en ello. Simplemente hay que hacer dos cambios respecto como pasaríamos un parámetro simple:

1) En la cabecera del proceso/función, hemos de especificar que el parámetro es un vector escribiendo, en el lugar que le corresponde como parámetro que es, lo siguiente:

***tipo POINTER nombrevector***

donde "tipo" es el tipo de datos de los elementos del vector, "pointer" es una palabra que se ha de escribir tal cual y "nombrevector" es el nombre del vector que queramos ponerle para trabajar con él en el interior del código del proceso/función. A partir de ahí, en ese código utilizaremos el vector como normalmente.

Si estamos hablando de TDUs, se hace muy parecido. En la cabecera de la función hay que especificar que el parámetro de es un tipo personalizado escribiendo, en el lugar que le corresponde como parámetro que es, lo siguiente:

***tipoPropio POINTER variableDeEseTipo***

donde "tipoPropio" es el nombre del TDU, "pointer" es una palabra que se ha de escribir tal cual y "variableDeEseTipo" es el nombre de la variable de ese tipo de datos que utilizaremos dentro del código del proceso.

Fíjate que estamos hablando siempre de vectores o de TDUs, no de tablas multidimensionales. Para lograr pasar este tipo de datos por parámetro, el proceso es diferente y todavía un poco más "rebuscado" (repito que se necesitan tener conocimientos de gestión de memoria) con lo que no se explicará en este manual.

2) En la llamada al proceso/función, se ha de preceder con el símbolo "&" el nombre del vector (y no poner corchetes ni nada) que se pasa por parámetro o bien el de una variable cuyo tipo sea el TDU , así:

```
proceso(&mivector);
```

o

```
proceso(&mivariabledeuntipopersonalizado);
```

Veamos un ejemplo. Vamos a crear un proceso que reciba un vector por parámetro y se dedique y mostrar por pantalla el valor de sus elementos:

```
Import "mod_text";
Import "mod_key";

process main()
private
    string mitabla[3]="aswert","asd2","asdfa","ppaw";
end
begin
  miproceso(&mitabla);
end

process miproceso(string pointer lalala)
private
    int i;
end
begin
  for(i=0;i<4;i++)
      write(0,100,10+10*i ,4,lalala[i]);
  end
  repeat
      frame;
  until(key(_esc))
end
```

Un detalle muy importante: cuando pasamos vectores como parámetros, Bennu está diseñado de tal manera que cualquier cambio que hagamos dentro del código interno del proceso/función en cuestión a los valores de los elementos de ese vector, esos cambios permanecerán fuera de dicho proceso/función, con lo que si accedemos a esos elementos desde otro sitio, esos cambios se verán reflejados. Esto se puede ver fácilmente con este ejemplo:

```
Import "mod_text";
```

```
Import "mod_key";

process main()
private
   string mitabla[2]="Popo","Lela","Asdfa";
end
begin
  leervector(&mitabla,100);
  cambiarvector(&mitabla);
  leervector(&mitabla,200);
  repeat
     frame;
  until(key(_esc))
end

function leervector(string pointer lalala, int posx)
private
   int i;
end
begin
  for(i=0;i<3;i++)
      write(0,posx,10+10*i ,4,lalala[i]);
  end
  return;
end

function cambiarvector(string pointer lalala)
private
    int i;
end
begin
  for(i=0;i<3;i++)
      lalala[i]="CAMBIADO";
  end
  return;
end
```

Igual que hemos hecho con vectores, también podríamos utilizar tipos definidos por el usuario (TDU) como parámetros de funciones, con (otra vez) la particularidad de que cualquier modificación en los valores de sus campos efectuada dentro del cuerpo de la función se mantendrán una vez acabada la ejecución de ésta (en la continuación de la ejecución del programa). Un ejemplo:

```
Import "mod_text";
Import "mod_key";

type tipo
          int a;
          string b;
end

process main()
private
          tipo t;
end
begin
  t.a=1;
  t.b="a";
          hola(&t);
          write(0,100,100,4,t.a);
          write(0,100,110,4,t.b);
          repeat frame; until(key(_esc))
end

function hola(tipo pointer pepito)
begin
pepito.a = 2;
pepito.b = "b";
```

```
end
```

Verás que lo que aparecerá por pantalla serán los nuevos valores de los campos de la variable *t* -del TDU *tipo*-, una vez se haya ejecutado la función "hola()".

Si piensas un poco, verás que lo que hemos hecho en estos ejemplos, sin decirlo, es crear una función que devuelve no ya enteros o cadenas como hasta ahora -que también: la sentencia return se puede seguir usando- sino que es capaz de devolver otro tipo de valores, como vectores o TDUs, debido a que las modificaciones hechas internamente en la función se mantienen después. Esta manera de hacer se podría hacer extensible a otros tipos lo más complejos posibles, con lo que ya podríamos construir funciones que pudieran devolver cualquier tipo de datos.

## *Ordenar los elementos de un vector

En muchas ocasiones usaremos vectores simples, los valores de cuyos elementos (numéricos o de cadena) no estarán ordenados, y tendremos la necesidad de ordenar dichos elementos de menor a mayor en el caso de vectores numéricos o de la "a" a la "z" en el caso de vectores de cadenas. Es decir, que el valor del elemento vector[0] sea menor que el de vector[1], y éste que el de vector[2], y así. En muchas ocasiones también usaremos vectores de estructuras, y desearemos ordenar los registros según uno de sus campos, típicamente numérico, de tal manera que según el valor de dicho campo los registros aparezcan dispuestos correlativamente. Para lograr ambos objetivos, Bennu dispone de una familia de funciones especializada en ordenaciones. Son las funciones **sort()**, **ksort()** y **quicksort()**, cada una con sus particularidades, como ahora veremos,pero todas ellas definidas en el módulo "mod_sort".

**SORT(VECTOR)**

Esta función ordena los elementos de un vector -tabla unidimensional- de datos simples. Los elementos del vector pueden ser de cualquier tipo simple válido: entero, decimal, cadenas...La ordenación se efectuará numéricamente o alfabéticamente según el caso.

Esta función tiene un segundo parámetro opcional de tipo entero que indica el número de elementos del vector (a partir del primero) que se quieren ordenar, por si no se quisiera ordenar todo el vector entero

Esta función devolverá 0 si ha habido algún error.

Si se desea ordenar los registros de un vector de estructuras a partir de los valores de los elementos de uno de sus campos, se recomienda usar **ksort()**. Para una ordenación más avanzada, consultar la función **quicksort()**

PARAMETROS:

XXX VECTOR: Matriz unidimensional cuyos elementos (de cualquier tipo) se ordenarán
INT NUMELEM: Número de elementos del vector que se ordenarán -opcional-

Un ejemplo sencillo que ordena un vector de enteros:

```
import "mod_sort";
import "mod_text";
import "mod_key";

const
   NUMELEM=18;
end

process main()
private
   int i;
   int vector1[NUMELEM-1]=1,5,3,7,2,4,5,3;
end
begin
  write(0,10,10,3,"Vector1 sin ordenar");
  for(i=0;i<NUMELEM;i++)
          write(0,10,20+15*i,4,vector1[i]);
  end
```

```
  sort(vector1);

  write(0,150,10,3,"Vector1 ordenado");
  for(i=0;i<NUMELEM;i++)
          write(0,150,20+15*i,4,vector1[i]);
  end

  while(!key(_esc))
     frame;
  end
end
```

Fíjate que los elementos vacíos, al rellenar automáticamente con 0, se ordenan siempre al principio de todo, con lo que los valores del vector aparecen ordenados hacia el final.¿Cómo podríamos ordenar solamente los elementos con contenido, obviando aquellos que están vacíos? Precisamente la función **sort()** tiene un segundo parámetro opcional, que si se especifica, indica el número de elementos que deseas ordenar (si no se pone, se sobreentiende que son todos los elementos del vector). Ya que sabemos que los elementos con contenido son ocho, prueba a modificar del ejemplo la línea donde aparece la función **sort()** por esta otra:

```
sort(vector1,8);
```

Al ejecutar esta variante podrás comprobar como ahora ha ordenado los primeros ocho elementos que se ha encontrado en el vector desordenado -los únicos que tenían valor- y los ha colocado, ordenados ya, en los ocho primeros puestos del vector ordenado, dejando el resto de casillas hasta el final llenas de 0.

Parece que está todo solucionado, pero surge un problema. En este ejemplo sabemos cuántos elementos llenos tenemos en el vector desordenado, pero, normalmente esa información no la tendremos. ¿Cómo podemos hacer entonces para ordenar sólo los elementos con contenido sin saber cuántos hay? A continuación voy a proponer una solución, pero no será una solución válida para todos los ámbitos. Me explico: esta solución presupone que todos los elementos iguales a 0 son los elementos vacíos; en principio esto parece correcto, pero es muy frecuente que haya elementos "llenos" cuyo contenido sea precisamente 0, ya que el cero es un valor tan válido como otro cualquiera, por lo que estos elementos no se tendrían en cuenta a la hora de ordenar ya que serían considerados elementos vacíos, con el error que esto conlleva. Avisado estás. Bien, la solución que propongo es sencilla : nos generamos nosotros mismos una función que, pasándole como parámetros el vector en cuestión y el número de elementos totales, nos devolverá el número de elementos llenos -los elementos con valores diferentes de 0-.A esta función la llamaremos "count()".

Fíjate que he comentado que esta función recibe como parámetro un vector. Recuerda que pasar vectores como parámetros de procesos/funciones no es tan sencillo como pasar cualquier otro tipo de dato, y se requiere un cuidado a la hora de escribir la notación correcta. Si tienes dudas, consulta el apartado correspondiente en un capítulo anterior.

```
import "mod_sort";
import "mod_text";
import "mod_key";

const
   NUMELEM=18;
end

process main()
private
   int i, numero;
   int vector1[NUMELEM-1]=1,5,3,7,2,4,5,3;
end
begin
  write(0,10,10,3,"Vector1 sin ordenar");
  for(i=0;i<NUMELEM;i++)
          write(0,10,20+15*i,4,vector1[i]);
  end

  numero=count(&vector1,NUMELEM);
  sort(vector1,numero);
```

```
  write(0,150,10,3,"Vector1 ordenado");
  for(i=0;i<NUMELEM;i++)
          write(0,150,20+15*i,4,vector1[i]);
  end

  while(!key(_esc))
     frame;
  end
end

function count(int pointer matriz, int numero)
private
  int llenos=0;
  int j;
end
begin
  for(j=0;j<numero;j++)
          if(matriz[j]!=0)llenos++; end
  end
  return llenos;
end
```

También podemos comprobar que la función **sort()** funciona con vectores de cadenas:

```
import "mod_sort";
import "mod_text";
import "mod_key";

const
   NUMELEM=4;
end

process main()
private
   int i, numero;
   string vector1[NUMELEM-1]="Hola qué tal","Yo muy bien", "Me alegro", "Gracias.";
end
begin
  write(0,10,10,3,"Vector1 sin ordenar");
  for(i=0;i<NUMELEM;i++)
          write(0,10,20+15*i,0,vector1[i]);
  end

  sort(vector1);

  write(0,150,10,3,"Vector1 ordenado");
  for(i=0;i<NUMELEM;i++)
          write(0,150,20+15*i,0,vector1[i]);
  end

  while(!key(_esc))
     frame;
  end
end
```

| **KSORT(VECTOR,CAMPO)** |
|---|
| Esta función ordena los registros de un vector de estructuras, a partir de los valores de uno de sus campos,especificado éste como segundo parámetro (es lo que se llama la variable de ordenación).<br><br>Esta función tiene un tercer parámetro opcional de tipo entero que indica el número de elementos del vector (a partir del primero) que se quieren ordenar, por si no se quisiera ordenar todo el vector entero<br><br>Esta función devolverá 0 si ha habido algún error.<br><br>Si se desea ordenar un vector de datos simples (numéricos o de cadena) , se puede utilizar **sort()**. Para una ordenación más avanzada, consultar la función **quicksort()** |

PARAMETROS

XXX VECTOR: Vector de estructuras cuyos registros se ordenarán
XXX CAMPO: Nombre del campo de la estructura cuyos valores definirán el orden de los registros.
INT NUMELEM: Número de elementos del vector que se ordenarán -opcional-

Un ejemplo de su utilización podría ser éste:

```
import "mod_sort";
import "mod_text";
import "mod_key";
import "mod_video";

Type _jugador
    String nombre;
    int puntuacion;
End

Const
    maxjugadores = 5;
end

process main()
private
    _jugador jugador[maxjugadores-1];
    int i;
end
Begin
    set_mode(640,480);
    // Rellenamos la estructura con algunos registros
    jugador[0].nombre = "Jugador3";
    jugador[1].nombre = "Jugador2";
    jugador[2].nombre = "Jugador5";
    jugador[3].nombre = "Jugador4";
    jugador[4].nombre = "Jugador1";

    jugador[0].puntuacion = 70;
    jugador[1].puntuacion = 30;
    jugador[2].puntuacion = 80;
    jugador[3].puntuacion = 90;
    jugador[4].puntuacion = 50;

    //Mostramos el vector desordenado
    Write(0,100,10,4,"Vector desordenado");
    for(i=0; i<maxjugadores; i++)
        write(0,100,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
    end

    //Ordeno el vector utilizando el campo nombre como variable de ordenación
    ksort(jugador,jugador[0].nombre,maxjugadores);

    //Mostramos el vector ordenado por el nombre
    Write(0,300,10,4,"Vector ordenado por el nombre");
    for(i=0; i<maxjugadores; i++)
        write(0,300,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
    end

    //Ahora ordeno el vector utilizando el campo puntuación como variable de ordenación
    ksort(jugador,jugador[0].puntuación,maxjugadores);

    //Mostramos el vector ordenado por puntuación
    Write(0,500,10,4,"Vector ordenado por puntuación");
    for(i=0; i<maxjugadores; i++)
        write(0,500,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
    end
```

```
    Repeat
         frame;
    Until(key(_esc))

End
```

Fijarse que hemos utilizado en la función **ksort()** el tercer parámetro, opcional, para establecer el número de elementos que deseamos ordenar.

Como curiosidad comentar que la función **sort()** también es capaz de ordenar vectores de estructuras, pero siempre las ordenará siempre por su primer campo, ya que no se puede definir cuál es el campo que queremos que haga de variable de ordenación y por defecto siempre elige el primero sin posibilidad de cambiarlo. Para probar esto, puede añadir las siguientes líneas al ejemplo anterior, justo antes del bucle repeat/until del final:

```
    // Volvemos a ordenar por nombre, ya que nombre es el primer campo de cada registro
    sort(jugador);

    // Mostramos el vector ordenado por nombre
    Write(0,300,190,4,"Vector ordenado por el nombre");
    for(i=0; i<maxjugadores; i++)
        write(0,300,200 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
    end
```

Vemos que, efectivamente, ordena por nombre, pero con **sort()** es imposible en este ejemplo ordenar por puntuación, ya que esta campo no es el primero en los registros.

Otro programa que pone en práctica el uso de la función **ksort()** podría ser éste, fácil de entender:

```
import "mod_sort";
import "mod_text";
import "mod_key";

type struct1
   string strng;
   int intgr;
end

type struct2
   int intgr;
   string strng;
end

process main()
private
   struct1 st[6] = "s10",10,"s5",5,"s210",210,"s3",3,"s33",33,"s12",12,"s3",3;
   struct2 st2[6] = 10,"s10",5,"s5",210,"s210",3,"s3",33,"s33",12,"s12",3,"s3";
   struct1 stbak1[6];
   struct2 stbak2[6];
   int i;
end
begin
/*Ordenamos el vector st con el campo String (el primero de la estructura) como variable
de ordenación.Lo hacemos dos veces: una ordenando todos los elementos del vector y otra
sólo los cuatro primeros*/
write(0,0,0,0,"Orden por String (primer campo)");

stbak1 = st; /*Guardamos una copia de st en un vector auxiliar stbak1, para mantener la
(des)ordenación de los datos original*/
write(0,0,10,0,"Toda la lista");
ksort(st, st[0].strng);
/*Fijarse que para averiguar el número de elementos del vector se divide el tamaño total
del vector por el tamaño de uno de sus elementos. */
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,0,(2+i)*10,0,st[i ].intgr+" , "+st[i ].strng);
end
```

```
st=stbak1; //Volvemos a desordenar st tal como estaba al principio del programa
write(0,160,10,0,"Los primeros 4");
ksort(st, st[0].strng, 4);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,160,(2+i)*10,0,st[i ].intgr+" , "+st[i ].strng);
end

/*Ordenamos el vector st con el campo Int (el segundo de la estructura) como variable de
ordenación.Lo hacemos dos veces: una ordenando todos los elementos del vector y otra sólo
los cuatro primeros*/
write(0,0,100,0,"Orden por Int (segundo campo)");

st=stbak1;
write(0,0,110,0,"Toda la lista");
ksort(st, st[0].intgr);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,0,100+(2+i)*10,0,st[i ].intgr+" , "+st[i ].strng);
end

st=stbak1;
write(0,160,110,0,"Los primeros 4");
ksort(st, st[0].intgr, 4);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,160,100+(2+i)*10,0,st[i ].intgr+" , "+st[i ].strng);
end

write(0,0,190,0,"Presione ESC para el siguiente test");
while(!key(_ESC)) frame; end//Hasta que no se pulse ESC no se hace nada
while(key(_ESC)) frame; end //Mientras se esté pulsando tampoco
delete_text(0);

/*A continuación realizo las mismas cuatro operaciones (ordenar por el primer campo todos
los elementos, ordenar por el primer campo los 4 primeros elementos, ordenar por el
segundo campo todos los elementos y ordenar por el segundo campo los 4 primeros
elementos) pero efectuándolas sobre el vector st2*/
write(0,0,0,0,"Orden por Int (primer campo)");

stbak2=st2;
write(0,0,10,0,"Toda la lista");
ksort(st2, st2[0].intgr);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,0,(2+i)*10,0,st2[i ].intgr+" , "+st2[i ].strng);
end

st2 = stbak2;
write(0,160,10,0,"Los primeros 4");
ksort(st2, st2[0].intgr, 4);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,160,(2+i)*10,0,st2[i ].intgr+" , "+st2[i ].strng);
end

write(0,0,100,0,"Orden por String (segundo campo)");

st2 = stbak2;
write(0,0,110,0,"Toda la lista");
ksort(st2, st2[0].strng);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,0,100+(2+i)*10,0,st2[i ].intgr+" , "+st2[i ].strng);
end

st2 = stbak2;
write(0,160,110,0,"Los primeros 4");
ksort(st2, st2[0].strng, 4);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,160,100+(2+i)*10,0,st2[i ].intgr+" , "+st2[i ].strng);
end

while(!key(_ESC)) frame; end
```

```
end
```

| QUICKSORT(VECTOR,TAMAÑOELEM,NUMELEM,OFFSET,TAMAÑODATO,TIPODATO) |
|---|
| Esta función ordena un vector -tanto simples como de estructuras- utilizando el algoritmo matemático de ordenamiento QuickSort.<br><br>Esta función es muy útil para ordenar vectores de tipos definidos por el usuario (datos TYPE), en donde uno de sus campos actúe como la variable de ordenamiento (es decir, que sus valores sean los que se usarán para ordenar todos los diferentes datos de ese tipo). No obstante, una de las limitaciones de esta función es que la variable de ordenamiento no puede ser de tipo cadena.<br><br>Para vectores simples, puede usarse la función **sort()**. Para vectores de estructuras, puede usarse **ksort()**<br><br>Esta función devolverá 0 si ha habido algún error.<br><br>PARAMETROS<br>POINTER VECTOR: Puntero al primer elemento del vector que será ordenado.<br>INT TAMAÑOELEM: Tamaño de un elemento/registro del vector, en bytes.<br>INT NUMELEM: Número de elementos totales del vector<br>INT OFFSET : Número de bytes respecto el inicio de un elemento donde se sitúa el campo que ejerce de variable de ordenamiento.<br>BYTE TAMAÑODATO : Tamaño en bytes del campo que ejerce de variable de ordenamiento.Por ejemplo, float=4,int=4,word=2,byte=1.<br>BYTE TIPODATO: El tipo del campo que ejerce de variable de ordenamiento (0=entero,1=decimal) |

Un ejemplo:

```
import "mod_sort";
import "mod_text";
import "mod_key";
import "mod_video";

Type _jugador
    String nombre;
    int puntuacion;
End
Const
    maxjugadores = 5;
end

process main()
private
    _jugador jugador[maxjugadores-1];
    int i;
end
Begin
    set_mode(640,480,16);
    // Rellenamos la estructura con algunos registros
    jugador[0].nombre = "Jugador3";
    jugador[1].nombre = "Jugador2";
    jugador[2].nombre = "Jugador5";
    jugador[3].nombre = "Jugador4";
    jugador[4].nombre = "Jugador1";

    jugador[0].puntuacion = 70;
    jugador[1].puntuacion = 30;
    jugador[2].puntuacion = 80;
    jugador[3].puntuacion = 90;
    jugador[4].puntuacion = 50;

   //Mostramos el vector desordenado
    Write(0,100,10,4,"Vector desordenado");
    for(i=0; i<maxjugadores; i++)
        write(0,100,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
    end
```

```
/*IMPORTANTE: La función quicksort() no puede ser usada para ordenar Strings, (ya que una
cadena en Fénix es en realidad un puntero a la cadena real, con lo que quicksort()
ordenaría las direcciones de memoria). Esto implica que en este ejemplo no se pueda
ordenar por el campo nombre.*/

    // Ordeno por el campo puntuacion.
    quicksort(&jugador[0],sizeof(_jugador),maxjugadores,sizeof(String),sizeof(int),0);

   //Mostramos el vector ordenado
   Write(0,300,10,4,"Vector ordenado");
    for(i=0; i<maxjugadores; i++)
        write(0,300,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
    end

    Repeat
         frame;
    Until(key(_esc))

End
```

Fijarse en los valores que tienen los parámetros de la función **quicksort()**. El primero hemos dicho que es un puntero al primer elemento del vector que queremos que se ordene: como queremos ordenar todo el vector, el primer elemento que queremos que se ordene será el primer elemento del vector, así que su puntero será *&jugador[0]*. El segundo parámetro ha de valer el tamaño en bytes de un registro del vector; como este vector es de elementos de tipo *_jugador*, para averiguar el tamaño en bytes de un elemento de este tipo hacemos uso de la funcion **sizeof()** así: *sizeof(_jugador)*. El tercer elemento es el número de elementos totales del vector: *maxjugadores*. El cuarto es el número de bytes que hay entre el principio de cada registro y la posición dentro de éste que ocupa la variable de ordenamiento. Como bien se comenta en el código, la función **quicksort()** es incapaz de utilizar variables de ordenamiento que sean de tipo cadena, así que el campo "nombre" del tipo *_jugador* no lo podremos utilizar para ordenar, pero sí el campo "puntuacion". Este campo es el segundo de la estructura, y por lo tanto hay una serie de bytes entre el principio de ésta y este campo. En concreto, esta serie de bytes viene dada por lo que ocupen el conjunto de campos que haya antes de "puntuacion", y en nuestro caso, sólo es uno, "nombre", de tipo cadena. Así pues, para saber cuanto ocupa este campo "nombre" que nos dará la diferencia de bytes entre el principio de la estructura y la posición del campo "puntuacion", usaremos como antes la función **sizeof()** así: *sizeof(string)*. El quinto elemento es el tamaño en bytes del campo que será la variable de ordenación. El campo "puntuacion" es un Int, así que pondremos *sizeof(int)*. Y el sexto elemento indica el tipo de campo que es la variable de ordenación; como "puntuacion" es entera, pondremos un 0. (Este último parámetro sirve para distinguir los Int de los Float, ya que ambos ocupan 4 bytes en memoria y no se pueden distinguir utilizando el quinto parámetro).

Otro programa que pone en práctica el uso de la función **quicksort()** podría ser éste, fácil de entender:

```
import "mod_sort";
import "mod_text";
import "mod_key";

type struct1
  int intgr;
  string strng;
end;

type struct2
  string strng;
  int intgr;
end;

process main()
private
   int array[13]=33,42,12,3,92,34,11,99,66,44,42,55,1,20;
   float flt[]=33.0, 42.2, 12.21, 3;
   int i;
   struct1 st[6]=10,"s10",5,"s5",210,"s210",3,"s3",33,"s33",12,"s12",3,"s3";
   struct2 st2[6]="s10",10,"s5",5,"s210",210,"s3",3,"s33",33,"s12",12,"s3",3;
end
```

```
begin
//Ordenamos el vector de enteros.
/*Fijarse en los valores de los parámetros de la función quicksort(): para averiguar el
número de elementos del vector se divide el tamaño total del vector por el tamaño de uno
de sus elementos. El offset es 0 porque es un vector simple. Los demás valores son
fácilmente deducibles.*/
quicksort(&array[0], sizeof(array[0]), sizeof(array)/sizeof(array[0]), 0,
sizeof(array[0]), 0);
write(0,0,0,0,"Orden array enteros");
for(i=0;i<sizeof(array)/sizeof(array[0]);i++)
write(0,0,10+i*10,0,array[ i]);
end

//Ordenamos el vector de decimales
quicksort(&flt[0], sizeof(flt[0]), sizeof(flt)/sizeof(flt[0]), 0, sizeof(flt[0]), 1);
write(0,160,0,0,"Orden array floats");
for(i=0;i<sizeof(flt)/sizeof(flt[0]);i++)
write(0,160,10+i*10,0,flt[ i]);
end

write(0,0,190,0,"Presione ESC para el siguiente test");
while(!key(_ESC)) frame; end //Hasta que no se pulse ESC no se hace nada
while(key(_ESC)) frame; end //Mientras se esté pulsando tampoco
delete_text(0);

//Ordenamos el vector st, que tiene el campo entero el primero de cada registro
write(0,0,0,0,"Orden por Int (offset 0)");
quicksort(&st, sizeof(st[0]), sizeof(st)/sizeof(st[0]), 0, sizeof(st[0].intgr), 0);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,0,10+i*10,0,st[i ].intgr+" , "+st[i ].strng);
end

/*Ordenamos el vector st2, que tiene el campo entero el segundo de cada registro, después
de un campo de tipo cadena. Recordemos que quicksort() no es capaz de utilizar variables
de ordenamiento que sean de tipo cadena.*/
write(0,160,0,0,"Orden por Int (offset 4)");
quicksort(&st2, sizeof(st2[0]), sizeof(st2)/sizeof(st2[0]), 4, sizeof(st2[0].intgr), 0);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,160,10+i*10,0,st2[i ].intgr+" , "+st2[ i].strng);
end

while(!key(_ESC)) frame; end

end
```

# CAPITULO 6

## Tutoriales básicos: el Juego del Laberinto y un Ping-Pong

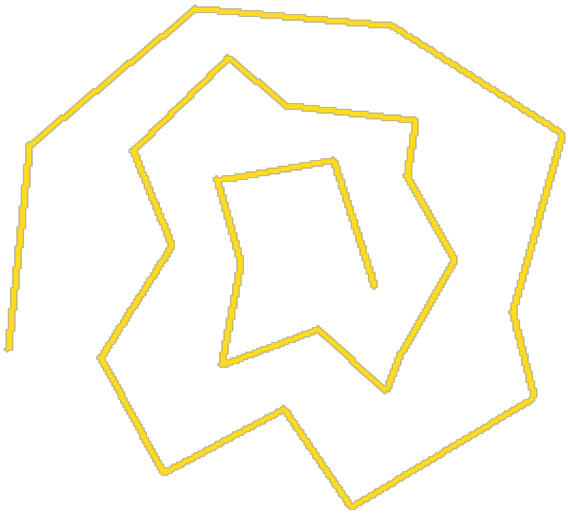
En este capítulo realizaremos paso a paso nuestros dos primeros juegos. Empezaremos por el "Juego del Laberinto", un juego donde hay que conseguir salir de un laberinto mediante los cursores del teclado, y seguidamente desarrollaremos un ping-pong.

### *Imágenes para el juego del laberinto

Lo primero que vamos a hacer es crearnos un fichero FPG –llamado “graficos.fpg”- con dos imágenes PNG. Una será de 30x30 píxeles y tendrá el código 001: representará el personaje que moveremos nosotros con el cursor. La otra representará el laberinto; será de 640x480 –la pantalla completa- y tendrá el código 002.

La imagen 001 puede ser algo parecido a esto (recuerda establecer correctamente las zonas transparentes):

y la imagen 002 –a escala-  a esto:



También podrías añadir alguna imagen de fondo visible por debajo del laberinto mediante **put_screen()** ó similar, para no ver la pantalla negra, pero esto es opcional.

### *El juego del laberinto básico. La orden "Collision()"

Bueno, empecemos. Vamos a aprovechar el primer ejemplo de código que escribimos en el capítulo 4 para empezar a escribir nuestro juego. Aquí lo vuelvo a escribir para recordarlo.

```
Import "mod_video";
Import "mod_map";
Import "mod_key";

GLOBAL
    INT id1;
END

PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
END

PROCESS personaje()
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        IF (key(_down)) y=y+10; END
        IF (key(_left)) x=x-10; END
        IF (key(_right)) x=x+10; END
        IF (key(_esc)) break; END
        FRAME;
    END
END
```

Con este código, recordarás que lo que teníamos es nuestro personaje que se puede mover por cualquier sitio de nuestra pantalla, incluso salir de ella. Ahora vamos a añadir el laberinto, el cual, será otro proceso. Recuerda que para hacer esto, hay que hacer dos cosas. Primero modifica el programa principal así:

```
PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
    laberinto();
END
```

Fíjate que lo único que hemos hecho ha sido añadir después de la creación del personaje, la llamada al proceso “laberinto()” para que se cree y visualice el laberinto. Está claro que lo segundo que tendremos que hacer es codificar este nuevo proceso “laberinto()”. Para eso, después del END que cierra el código del proceso “personaje()” , escribe lo siguiente:

```
PROCESS laberinto()
BEGIN
     x=320;y=240;graph=2;
     LOOP
             frame;
      END
END
```

Con esto lo único que hacemos es visualizar el laberinto en el centro de la pantalla: (establecemos valores a las variables ***X,Y*** y ***GRAPH*** y nos metemos en un bucle infinito para ir mostrando siempre la imagen), pero nuestro personaje, -el cual ha de aparecer en el centro del laberinto según las X e Y que hemos puesto-, se puede seguir moviendo igual.

Lo que vamos a hacer ahora es obligar a que si el personaje se mueve y choca contra el laberinto, vuelva al centro de la pantalla. Para ello, hemos de modificar el proceso “personaje()” de la siguiente manera:

```
PROCESS personaje()
BEGIN
    x=320; y=240; file=id1; graph=1;
```

```
    LOOP
        IF (key(_up)) y=y-10; END
        IF (key(_down)) y=y+10; END
        IF (key(_left)) x=x-10; END
        IF (key(_right)) x=x-10; END
        IF (collision(TYPE laberinto)) x=320;y=240; END
        FRAME;
    END
END
```

Fíjate que sólo hemos añadido una línea más a las que había: la línea *IF (collision(TYPE laberinto)) x=320;y=240; END*, donde nos encontramos con una nueva orden: *collision(TYPE laberinto);* (tendremos además que incluir el módulo "mod_grproc", que es donde esta orden está definida). Esta sentencia devuelve cierto si el gráfico del proceso que la ejecuta -o sea, "personaje()"- y el gráfico de alguna instancia cualquiera del proceso cuyo nombre se escribe entre paréntesis seguido de TYPE -o sea, "laberinto()"- chocan, ("colisionan") de alguna manera. Por lo tanto, el IF es cierto y se ejecutan las sentencias del interior. Y ahí nos encontramos dos órdenes, que vuelven a poner el proceso personaje en el centro de la pantalla. ¡Ya hemos hecho nuestro primer juego!

Al escribir la palabra TYPE dentro de los paréntesis de la orden **collision()** lo que estamos intentando detectar son, insisto, todas las colisiones que tenga el proceso "personaje()" con TODAS las instancias del proceso "laberinto()" que estén activas en este momento. Es decir, que si hubiéramos hecho varias llamadas dentro de nuestro código al proceso "laberinto()", tendríamos por cada una de ellas una instancia de él (se verían múltiples laberintos en nuestro juego), y con la palabra TYPE lo que hacemos es detectar las colisiones con cualquiera de ellas. Evidentemente, en este juego no sería necesario utilizar TYPE porque sólo tenemos una instancia de "laberinto()" funcionando (o sea, sólo hemos creado el proceso "laberinto()" una vez), pero funciona igual. De todas maneras, alguna vez nos encontraremos con la necesidad de detectar la colisión de nuestro personaje con una instancia CONCRETA de un proceso (como por ejemplo, un enemigo concreto de entre tantos creados a partir de un único proceso "enemigo()"). ¿Cómo lo haríamos? En ese caso, el parámetro de **collision()** debería ser el ID de esa instancia concreta con la que se desea detectar si hay colisión o no. Es decir, resumiendo, podemos utilizar la orden **collision()** de dos formas. O bien:

```
collision(TYPE nombreProceso);
```

donde detectamos las colisiones del proceso actual con TODAS las instancias de un proceso concreto dado por "nombreProceso()", o bien:

```
collision(idProceso);
```

donde detectamos las colisiones del proceso actual sólo con aquella instancia que tenga un identificador dado por idProceso.

Por otro lado, hay que aclarar que el caso de utilizar **collision()** de la manera *collision(TYPE nombreProceso);* en realidad, **collision()** no devuelve “verdadero” o "falso" simplemente. Lo que devuelve en realidad es el ID del proceso con el que colisiona. Como un ID válido (es decir, correspondiente a un proceso activo existente) es siempre mayor que 0, y Bennu entiende que cualquier valor diferente de 0 es tomado como "verdadero", nos aprovechamos de este hecho para utilizar **collision()** de esta manera: si devuelve 0 es que no ha habido colisión, y si devuelve otra cosa, sí que ha habido, y en realidad esa "otra cosa" es el ID del proceso con el que se ha colisionado.Esto es importante porque aunque ahora no lo necesitemos, más adelante veremos que nos interesará recoger este ID para saber con qué proceso hemos colisionado en concreto.En el caso de utilizar **collision()** en el modo *collision(idProceso);,* cuando existe colisión no se devuelve *idProceso*, sino simplemente 1.

Y otro detalle más: **collision()** se comprueba en cada frame del proceso actual, pero puede ocurrir que antes de llegar a él, varios procesos hayan colisionado con el proceso actual. En ese caso, **collision()** sólo devolverá el ID del último proceso con el que ha colisionado antes del fotograma, perdiéndose los otros ID anteriores. Esto se podría evitar, por ejemplo, usando un bucle y un vector, del tipo:

```
i=0;
while (vectorids[i]=collision (type otroProc))
/*Vectorids[i] guarda el identificador de la instancia concreta de otroProc con la que se
ha colisionado, las cuales pueden ser varias a la vez. Cuando se haya acabado de
registrar todas esas colisiones, se habrán rellenado unos determinados elementos de dicho
vector y el bucle se termina porque la condición pasa a ser 0 -falsa-, (que es lo que
devuelve collision cuando no se detecta colisiones). Justo después de este bucle se puede
```

```
proseguir con el código normal del juego pudiendo utilizar todos los elementos de
Vectoids.*/
            i++;
end
```

Fijarse finalmente que en este ejemplo, si quisiéramos hacer que nuestro protagonista se moviera más despacio o más deprisa tendríamos que alterar las cuatro cifras. Si en vez de 10 hubiésemos creado en la sección de declaraciones del programa principal una constante llamada *velocidad*, por ejemplo, bastaría con alterar el valor de la constante al principio del programa para alterar todo el movimiento, lo que en un programa extenso nos ahorraría muchísimo tiempo y nos aseguraría que no nos hemos dejado nada por cambiar.

Tampoco sería mala idea crear dos constantes más, llamadas por ejemplo *reshor* y *resver* (de "resolución horizontal" y "resolución vertical", respectivamente, y asignarles el valor de 640 y 480. De esta manera, cuando usemos la orden **set_mode()**, podríamos hacer *set_mode(RESHOR,RESVER,32);* y cuando queramos situar al gráfico del laberinto y al personaje en el centro de la pantalla simplemente tendríamos que asignar a sus variables ***X*** e ***Y*** los valores RESHOR/2 y RESVER/2 respectivamente. Con esto ganamos que, si algún día decidiéramos cambiar la resolución de la pantalla del juego por otra (por ejemplo 800x600), no tuviéramos que cambiar "a mano" todas los valores numéricos de las coordenadas de los diferentes procesos (es decir, el 320 que vale la X por el 400 y el 240 de la Y por el 300): simplemente tendríamos que cambiar los valores de las dos constantes y automáticamente todo quedaría ajustado sin hacer nada más

## *Añadir giros al movimiento del personaje. La orden "Advance()" y la variable local predefinida ANGLE

Ahora veamos el siguiente caso: avanzar y girar. Lo primero que hay que saber es que cuando creamos un gráfico en Bennu siempre estará "mirando" hacia la derecha. Por lo tanto, es buena idea que, al dibujar objetos vistos desde arriba la parte de delante, éstos estén mirando hacia la derecha. Así nos evitaremos muchos problemas (fijaros que yo el triángulo lo he dibujado hacia la derecha). Para girar nuestro triángulo y hacer que “mire” hacia otro sitio habría que sustituir los cuatro IF de movimiento por estos otros:

```
IF (key(_up)) advance(10); END
IF (key(_down)) advance(-10); END
IF (key(_left)) angle=angle+7500; END
IF (key(_right)) angle=angle-7500; END
```

Como veis es muy simple. La sentencia **advance()**, definida en el módulo "mod_grproc", hace avanzar el gráfico el número de pixeles indicado entre paréntesis en el sentido en el que está "mirando" el gráfico. Si el número es negativo lo hace retroceder

***ANGLE*** es una variable local predefinida, y representa el ángulo que existe entre una flecha imaginaria apuntando hacia la derecha hasta la orientación actual donde "mira" el proceso correspondiente. Esta variable se mide en milésimas de grado y va en sentido contrario a las agujas del reloj. Por lo tanto si ponemos *angle=90000;* el proceso estará mirando hacia arriba, *angle=180000;* mirará hacia la izquierda, *angle=270000;* mirará hacia abajo y *angle=360000;* o *angle=0;* hacia la derecha.

Así pues, si apretamos el cursor de arriba o abajo veremos que lo que hace el triángulo es desplazarse 10 píxeles en la dirección en la que está “mirando” en ese momento, y si apretamos derecha o izquierda el triángulo no se mueve de donde está pero sí que varía su orientación (gira) para un lado o para el otro.

## *Incluir múltiples enemigos diferentes. La variable local predefinida SIZE

¿Y si ahora quisieramos incluir enemigos que dificulten nuestro juego? La idea es que, además de no poder chocar contra el laberinto, nuestro personaje evite chocar contra enemigos que irán surgiendo al azar por la pantalla. Sabemos que para que aparezca un sólo enemigo sería suficiente con crear un proceso llamada, por ejemplo, "enemigo();" y hacer una llamada a ese proceso. ¿Pero y si queremos crear más de un enemigo? Basta con hacer más de

una llamada. Así:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_grproc";

GLOBAL
    INT id1;
END

PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
    laberinto();
    loop
        enemigo();
        frame;
    end
END
```

Fíjate: hemos puesto un LOOP en el proceso principal y dentro una llamada al proceso "enemigo()". Además, hemos escrito –antes no estaba- la sentencia frame dentro del bucle porque recordad que si no aparece una sentencia FRAME en un proceso con un bucle infinito, el programa se puede colgar (porque los demás procesos estarán pausados siempre hasta que el que no tiene frame llegue a él, y si está en un bucle infinito sin frame, eso nunca ocurrirá).

Tal como lo hemos hecho, en cada frame se creará un enemigo. Pero como por defecto el programa funciona a 25 frames por segundo (fps, o sea, imágenes que muestra en un segundo; y que recordad que se puede cambiar con la sentencia **set_fps()**) se crean demasiados enemigos -¡25 por segundo!-. Por lo tanto hay que hacer algo para que aparezcan menos. Para solucionar eso vamos a usar la conocida sentencia **rand()** dentro del proceso principal. Así:

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_grproc";
Import "mod_rand";

GLOBAL
    INT id1;
END

PROCESS Main()
BEGIN
    set_mode(640,480,32);
    id1=load_fpg("graficos.fpg");
    personaje();
    laberinto();
    loop
        if(rand(1,100)==1)
              enemigo();
        end
        frame;
    end
END
```

Esto haría que en uno de cada 100 frames se creara un enemigo, porque hay una probabilidad de 1 contra 100 de que el valor devuelto de rand sea igual a 1, y que por tanto se ejecute el interior del IF. Como es aleatorio a veces serán más, a veces serán menos. Pero es muy poco probable que aparezcan todos a la vez, o incluso más de uno o dos cada segundo. Naturalmente, podéis poner la frecuencia que os de la gana. Ej.: *IF (rand(1,100)<11)*. Está claro por ejemplo que si pusiéramos *IF(rand(1,100)<=100)* no estaríamos haciendo nada de nada,¿cierto?.

Ahora toca escribir el código del proceso “enemigo()”. Pero antes, fíjate que llamando tal cual al proceso “enemigo()” así a secas, todos los enemigos serán iguales. La manera más práctica –hay otras que se te pueden ocurrir- para solucionar esto es utilizar parámetros. Los parámetros recuerda que son datos que le pasamos al proceso en el momento de crearlo, y que se ponen entre los paréntesis al final de la llamada al proceso; luego, en la cabecera del código del proceso en si, debemos poner qué es cada dato. Por ejemplo, -sólo lo estoy recordando: esto ya lo vimos-, en la llamada al proceso "personaje()" podríamos haber puesto *personaje(320,240,1);* , y en la cabecera de su proceso *PROCESS personaje(x,y,graph)*, omitiendo entonces en su código las asignaciones de los valores ***X ,Y, GRAPH***. De esta manera le hemos asignado directamente en la llamada al proceso esos valores a esas variables (el primer valor a la primera variable, el segundo valor a la segunda variable,...). Por lo tanto, podríamos hacer lo mismo con el proceso "enemigo()" , siendo así muy fácil modificarlos y hacer que un enemigo salga, por ejemplo, en un lugar concreto de la pantalla. Vamos allí: el código del enemigo, suponiendo que los enemigos aparecen en la parte superior de la pantalla y que van hacia abajo, podría ser algo parecido a esto:

```
PROCESS enemigo(x,y,graph,size,int incrementox, int incrementoy)
BEGIN
    LOOP
                x=x+incrementox;
                y=y+incrementoy;
                IF (x<-20) BREAK; END
                IF (x>660) BREAK; END
                IF (y>500) BREAK; END
                FRAME;
    END
END
```

Y la llamada al proceso desde el programa principal quedaría finalmente:

```
enemigo(rand(1,640),-50,3,rand(50,150),rand(-10,10),rand(5,10));
```

De esta manera se crearía un enemigo en una *X* al azar entre 1 y 640, 50 puntos por encima de la pantalla, con gráfico 3 (habrá que crear un gráfico y meterlo en el fpg), con tamaño aleatorio (***SIZE*** es una variable local predefinida que indica el tamaño del gráfico con respecto al original, en tanto por ciento: 50 quiere decir la mitad y 200 el doble) y cantidad de movimiento horizontal y vertical aleatorio (variables estas dos últimas creadas por nosotros: no son predefinidas). Con estos datos es muy poco probable que se creen dos enemigos exactamente iguales.

Los tres IF que se han escrito son por motivos prácticos. Que el gráfico salga de la pantalla no quiere decir que el proceso haya terminado Por lo tanto seguirá ejecutándo el código para siempre (o hasta que se termine el programa, una de dos). Puede no parecer importante, pero hasta el ordenador más rápido terminará por dar saltos cuando tenga que manejar demasiados procesos, puesto que se están creando nuevos procesos continuamente. Por eso al proceso se le obliga a terminar cuando sale de la pantalla. Fíjate que aparece un BREAK, el cual fuerza al programa a romper el bucle y continuar justo después del END del mismo. En este caso, como después del END del LOOP viene el END del BEGIN (es decir, no hay más órdenes que ejecutar) el proceso finaliza.

Utilizando parámetros de esta manera, es mucho más fácil crear procesos similares pero con características diferentes. Imagínate que quieres en algún momento queréis crear un enemigo con estas características: x=100; , size=150; , incrementox=10; , incrementoy=5; y un gráfico diferente (graph=4:habría que meter ese gráfico también). Con los parámetros sería suficiente con llamar a ese proceso pasándole esos datos. Sin los parámetros habría que crear un nuevo proceso con su propio código (en el que la mayoría sería idéntico al del otro proceso, pero bueno). Como ves los parámetros son muy útiles.

Pero por ahora los enemigos son inofensivos. Lo único que hacen es aparecer en la pantalla, darse un paseíto y desaparecer. Ahora los vamos a convertir en un peligro para el personaje. En el proceso “personaje()” , después de la comprobación de colisión con el laberinto, habría que hacer una comprobación de colisión con los enemigos. El código a añadir en su proceso sería el siguiente:

```
IF (collision(TYPE enemigo)) BREAK; END
```

De esta manera, al chocar con algún enemigo cualquiera, la sentencia BREAK; rompería el bucle LOOP, continuando después del END y terminando el proceso (puesto que después se llega al END del proceso), y por tanto, haciendo desaparecer el protagonista y no teniendo pues el jugador nada que controlar, aunque el programa principal seguirá ejecutándose.

Vamos a jugar con ese BREAK;. Si después del END del LOOP escribimos alguna sentencia, ésta se cumplirá una vez que el personaje choque con algún enemigo, y antes de finalizarse. Por lo tanto podríamos poner algún comando que, por ejemplo, finalice el programa principal. De esta manera, cuando finalice el proceso "personaje()", éste hará finalizar también el proceso principal. La sentencia para ello, ya la vimos, es **exit()**.

## *Añadir explosiones

Aunque la verdad es que eso de salir así, sin más, queda un poco soso. Vamos a hacer que el personaje explote. Para eso crearemos otro proceso, llamado "explosion()" . La llamada la haremos después del LOOP del proceso "personaje()" ya que queremos mostrar la explosión justo después de que un enemigo cualquiera haya colisionado con el personaje y justo antes de que el proceso del personaje esté a punto de finalizar. Además, quitaremos de allí el **exit()** que acabamos de poner, porque si no no podríamos ver la explosión, ya que el programa principal acabaría ipsofacto sin dar tiempo a nada más. Por lo que de momento volveremos a la situación de que después de ver la explosión el juego continuará pero sin ningún personaje al que controlar.

Como queremos que la explosión suceda en el lugar donde está el personaje habrá que pasarle como parámetros la ***X*** y la ***Y*** del personaje, para que se conviertan en la ***X*** y la ***Y*** de la explosión. ¿Y cómo se hace eso? Muy sencillo. Supongamos que desde el "proceso_1()" se llama a "proceso_2()". En general, si en la llamada que hacemos desde "proceso_1()" a "proceso_2()" ponemos como parámetro el nombre de alguna de las variables de "proceso_1()", (no el valor como habíamos hecho hasta ahora) éste le pasará el valor actual de esa variable a la variable de "proceso_2()" que ocupe la posición correspondiente dentro de su cabecera. Por lo que si desde "personaje()" hacemos la llamada así: *explosion(x,y);* le estamos pasando el valor actual de ***X*** e ***Y*** del proceso "personaje()" a los dos parámetros de "explosion()" , y si en la cabecera de "explosion()" estos dos parámetros precisamente se llaman ***X*** e ***Y***,estaremos dándole los valores de ***X*** e ***Y*** del proceso personaje a los valores ***X*** e ***Y*** del proceso explosión, y por lo tanto la explosión tendrá lugar en el sitio donde está el personaje. Justo lo que queríamos. El código de "personaje()" queda así finalmente:

```
PROCESS personaje()
BEGIN
     x=600; y=240; graph=1;
     LOOP
             IF (key(_up)) advance(10); END
             IF (key(_down)) advance(-10); END
             IF (key(_left)) angle=angle+7500; END
             IF (key(_right)) angle=angle-7500; END
             IF (collision(TYPE laberinto)) x=320; y=240; END
             IF (collision(TYPE enemigo)) BREAK; END
             FRAME;
     END
     explosion(x,y);
END
```

Y el nuevo proceso "explosion()" es:

```
PROCESS explosion(x,y)
BEGIN
     FROM graph=11 TO 20; FRAME; END
END
```

Así de simple. Le hemos pasado como parámetros las coordenadas ***X*** e ***Y*** del proceso "personaje()", que es el proceso donde se llamará a "explosion()".

Habrás visto que usamos es una sentencia FROM (FOR también vale). Fíjate que como contador se ha puesto la variable ***GRAPH***, por lo que a cada iteración el gráfico asociado al proceso "explosion()" será diferente. Como a cada iteración hemos puesto que se ejecute la sentencia FRAME a cada iteración se visualizará el cambio de gráfico en pantalla, por lo que se creará un efecto donde parecerá que haya una explosión –si las imágenes las has dibujado que parezcan de fuego y que vayan creciendo en tamaño-, ya que se verán 10 imágenes rápidamente seguidas en el mismo sitio. Éste es un truco muy eficaz, pero depende de lo bueno que seas dibujando los momentos diferentes

de la explosión. En el ejemplo he usado diez imágenes, y he empezado a contar desde el 11 por una cuestión de orden, nada más: para dejar las imágenes 4-10 para otros usos todavía no especificados (los códigos de los FPGs no hace falta que sean correlativos: puede haber "saltos" entre ellos).

También se podría aplicar el mismo método de las explosiones a las colisiones con el laberinto (cambiando el reseteo de los valores de ***X*** e ***Y*** del proceso "personaje()" por una sentencia BREAK; y añadiendo la invocación al proceso "explosion()" u otro similar).

Ahora deberíamos decidir qué pasa cuando ya ha pasado la explosión: o se acaba el programa definitivamente, o podemos volver a empezar mostrando otra vez el protagonista intacto en el centro de la pantalla como si no hubiera pasado nada. En ambas opciones deberíamos modificar el proceso "explosion()".Veamos la primera. Ya vimos que tuvimos que quitar el **exit()** del proceso personaje porque no nos daba tiempo de ver la explosión. Pero esto no ocurre si ponemos el **exit()** después del FROM de "explosion()", así:

```
PROCESS explosion(x,y)
BEGIN
     FROM graph=11 TO 20; FRAME; END
     exit();
END
```

¿Por qué? Porque si ponemos el **exit()** dentro de "explosion()", primero se hará el FROM entero, y una vez que acabe –cuando ya hayamos visto toda la explosión- entonces se dice al programa principal que finalice. De la manera anterior, metiendo **exit()** en el proceso "personaje()", cuando se llegaba a la línea de llamada de "explosion()" ésta comenzaba a ejecutarse, pero "personaje()" por su lado también sigue ejecutándose línea a línea –recordad el tema de la concurrencia y la multitarea-, de manera que cuando "explosion()" iniciaba su ejecución, "personaje()" continuaba con la línea siguiente, que era el **exit()**, por lo que el programa se acababa directamente sin tener la posibilidad de verse la explosión completa.

La otra alternativa, volver a empezar, se realiza modificando el proceso "explosion()" así:

```
PROCESS explosion(x,y)
BEGIN
     FROM graph=11 TO 20; FRAME; END
     personaje();
END
```

Fíjate que lo que hacemos es una nueva llamada, una vez se ha visto la explosión, al proceso "personaje()", con lo que se vuelve a ejecutar el código de ese proceso, que define las variables ***X,Y*** y ***GRAPH***, y los movimientos de nuestro protagonista. Fíjate también que cuando "explosion()" llama a "personaje()" para crearlo de nuevo, el proceso "personaje()" anterior hace tiempo que dejó de existir –y por eso sólo se verá un personaje cada vez y no más-, ya que justamente cuando desde "personaje()" se llama a "explosion()", a "personaje()" se le acaba el código y muere. Por lo que digamos que lo que ocurre es que "personaje()" justo antes de morir crea a "explosion()", la cual, a su vez, justo antes de morir, vuelve a crear a "personaje()", con lo que se convierte en un bucle infinito de creación/destrucción de los procesos "personaje()"/"explosion()".

## *ANGLE, SIZE y otras variables locales predefinidas: SIZE_X, SIZE_Y, ALPHA y FLAGS

Ya hemos acabado nuestro juego del laberinto. No obstante, no está de más comentar con más calma y profundidad las posibilidades que nos ofrecen las nuevas variables introducidas en párrafos anteriores (***ANGLE*** y ***SIZE***), y de paso introducir también el conocimiento y uso de otras variables locales predefinidas muy interesantes , que se utilizarán extensamente a partir de ahora en este manual. Se trata de las variables ***SIZE_X***, ***SIZE_Y*** , ***ALPHA***, ***ALPHA_STEPS, FLAGS*** y ***XGRAPH***.

Miremos primero un ejemplo donde se insiste en el uso de ***ANGLE*** y ***SIZE***. Utilizaremos para ello el mismo archivo "graficos.fpg" del juego del laberinto (en concreto, su gráfico 001, el triángulo). Lo que hace este programa es cambiar la orientación (el ángulo) y el tamaño del gráfico en forma de triángulo según pulsemos la tecla adecuada del cursor del teclado. Lee los comentarios del código.

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";

Global
        int fpg;
End

process main()
Private
        int rzarrowid;
End
Begin
        fpg=load_fpg("graficos.fpg");
/*Creamos un nuevo proceso llamado "flecha" y almacenamos su ID en la variable global
"rzarrowid", para que podamos modificar su ángulo y su tamaño posteriormente*/
        rzarrowid = flecha();
//Escribimos el ángulo actualizado del proceso "flecha"
        write_var(0,60,0,0,rzarrowid.angle);
//Escribimos el tamaño actualizado del proceso "flecha"
        write_var(0,60,10,0,rzarrowid.size);
        Loop
//Incrementamos el ángulo en 2 grados
            if(key(_left)) rzarrowid.angle=rzarrowid.angle+2000; end
//Decrementamos el ángulo en 2 grados
            if(key(_right))rzarrowid.angle=rzarrowid.angle-2000; end
//Incrementamos el tamaño en un 10%
            if(key(_up))rzarrowid.size=rzarrowid.size+10; end
//Decrementamos el tamaño en un 10%
            if(key(_down))rzarrowid.size=rzarrowid.size-10;end
            If(key(_esc)) exit();End;
            Frame;
        End
End

/*Este proceso mostrará el triángulo en la pantalla. Modificando su variable ANGLE o SIZE
podremos rotarlo o hacerle zoom. Esto se podría hacer desde dentro del propio proceso
(más fácil) o, como en este ejemplo, obteniendo el identificador del proceso en el
momento de su creaación y modificando esas variables desde el programa principal*/
Process flecha()
Begin
        x = 320;
        y = 240;
        file=fpg;
        graph = 1;
        Loop
            Frame;
        End
End
```

Si pruebas el programa,podrás ver los valores que cogen ambas variables a medida que pulsemos las diversas teclas del cursor. Fíjate que cuando se gira una vuelta entera una imagen, el valor de la variable ***ANGLE*** no se pone a 0 automáticamente sino que continúa creciendo más allá de las 360000 milésimas de grado. Y esto puede ser un lío, y a la larga un problema. Si queremos que a cada vuelta, la variable ***ANGLE*** vuelva a valer 0, el truco está en aplicarle al valor que queremos que no sobrepase nunca el límite, la operación módulo con la cantidad máxima a partir de la cual se produce el "reseteo". Es decir, en nuestro caso, 360000. Pruébalo: sustituye las dos líneas donde se modifica el valor de ***ANGLE*** del triángulo (cuando se pulsan los cursores izquierdo y derecho) por éstas otras:

```
if(key(_left))rzarrowid.angle=(rzarrowid.angle+2000)%360000; end
if(key(_right))rzarrowid.angle=(rzarrowid.angle-2000)%360000; end
```

Si vuelves a ejecutar el programa, verás que ahora el gráfico gira igual, todas las vueltas que quieras, pero cada vez que se acaba de girar una vuelta, el valor de ***ANGLE*** vuelve a resetearse a 0, con lo que su valor máximo siempre será 360000. Este efecto es resultado de aplicar la teoría de la matemática modular, que en general viene a decir que cualquier operación matemática que obtenga un resultado, si a éste se le aplica el módulo de un número

determinado, el resultado permanecerá invariable si es menor que ese número, pero si es mayor, el resultado pasará a valer la parte "sobrante". Es decir, (2*5+3)%15=13, pero (2*5+3)%10=3. Es otra manera (más práctica) de entender lo que llamamos el resto de una división.

Tal como se ha dicho antes, tenemos otras variables locales predefinidas aparte de las ya vistas que son importante que conozcas:

| | |
|---|---|
| ***SIZE_Y*** | Esta variable especifica el escalado vertical que se va a realizar sobre el gráfico. A diferencia de ***SIZE*** el escalado sólo se aplica al alto del gráfico, siendo el valor por defecto 100 (100% del alto original).<br>Cuando se especifica un valor distinto de 100 en esta variable o en ***SIZE_X*** , se utilizan siempre los valores de ***SIZE_X*** y ***SIZE_Y*** , despreciándose el valor de ***SIZE***. |
| ***SIZE_X*** | Esta variable especifica el escalado horizontal que se va a realizar sobre el gráfico. A diferencia de ***SIZE*** el escalado sólo se aplica al ancho del gráfico, siendo el valor por defecto 100 (100% del ancho original).<br>Cuando se especifica un valor distinto de 100 en esta variable o en ***SIZE_Y*** , se utilizan siempre los valores de ***SIZE_X*** y ***SIZE_Y*** , despreciándose el valor de ***SIZE***. |

Un ejemplo (donde utilizaremos el mismo FPG del juego del laberinto, "graficos.fpg", y mostraremos el gráfico del laberinto, ya que al ser más grande que el del triángulo, se podrá apreciar mejor el efecto) bastante similar al anterior sería éste:

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";
import "mod_video";

Global
        int fpg;
End

process main()
Private
        int idlab;
End
Begin
    set_mode(640,480,32);
    fpg=load_fpg("graficos.fpg");
/*Creamos un nuevo proceso llamado "laberinto" y almacenamos su ID en la variable global "idlab"*/
    idlab = laberinto();
//Escribimos el tamaño horizontal del gráfico en % respecto el original
    write_var(0,60,0,0,idlab.size_x);
//Escribimos el tamaño vertical del gráfico en % respecto el original
    write_var(0,60,10,0,idlab.size_y);
/*Para comprobar cómo SIZE no se tiene en cuenta cuando se utilizan SIZE_X ó SIZE_Y*/
    write_var(0,60,20,0,idlab.size);
    Loop
            if(key(_left))idlab.size_x=idlab.size_x+10; end
            if(key(_right))idlab.size_x=idlab.size_x-10; end
            if(key(_up))idlab.size_y=idlab.size_y+10; end
            if(key(_down))idlab.size_y=idlab.size_y-10;end
            if(key(_esc)) exit();end
            Frame;
    End
End

Process laberinto()
Begin
        x = 320;
        y = 240;
        file=fpg;
        graph = 2;
        Loop
```

```
            Frame;
        End
End
```

| | |
|---|---|
| ***ALPHA*** | Contiene un valor entre 0 y 255 para indicar el grado de transparencia que tiene el gráfico dado por ***GRAPH***, independientemente de si dicho gráfico ya tiene de por si algún grado de transparencia intrínseco. Si fuera éste el caso, el nivel de trasparencia aportado por esta variable se le añadiría de forma uniformey sumativa a la transparencia propia de la imagen.<br>La transparencia total viene dada por ALPHA=0 y la transparencia original de la imagen no se ve alterada con ALPHA=255<br>Hay que tener en cuenta, no obstante, que para que esta variable sea tenida en cuenta por Bennu, la variable ***FLAGS*** -vista a continuación- ha de valer 8 (o la suma de 8 más otros valores:8+1,8+2,8+2+1,etc). |
| ***ALPHA_STEPS*** | Ésta es una variable GLOBAL, íntimamente ligada a ***ALPHA***, indica los niveles de transparencia deseados. Contiene un valor entre 1 y 255 (por defecto, 16) para indicar la cantidad de niveles de transparencia que se van a definir. Dicho en otras palabras, indica el número de saltos visibles de transparencia que puede haber entre el valor 0 y 255 de ***ALPHA***. Por ejemplo, si ***ALPHA_STEPS*** vale 4, sólo habrá 4 cambios apreciables de transparencia a lo largo de todos los valores posibles de ***ALPHA***. Para saber cada cuántos valores de ***ALPHA*** se producirá el cambio de transparencia, se puede usar la fórmula 256/ALPHA_STEPS. Si, para ALPHA_STEPS=4, habrá 256/4=64 valores de ***ALPHA*** consecutivos que ofrecerán la misma transparencia hasta llegar a un salto. Es evidente que a mayor valor de ***ALPHA_STEPS***, mayor "sensibilidad" existe al cambio de valores de ***ALPHA***. Los casos extremos serían ALPHA_STEPS=1 (donde no se produciría ningún cambio de transparencia en el gráfico a pesar de cambiar el valor de su ***ALPHA***) y ALPHA_STEPS=255, donde cada cambio en una unidad de ***ALPHA*** se traduciría visualmente en un cambio de transparencia en el gráfico. |

Un ejemplo (donde utilizaremos el mismo FPG del juego del laberinto, "graficos.fpg", y mostraremos el gráfico del laberinto, ya que al ser más grande que el del triángulo, se podrá apreciar mejor el efecto) bastante similar al anterior sería éste:

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";
import "mod_video";

Global
        int fpg;
End

process main()
Private
        int idlab;
End
Begin
    fpg=load_fpg("graficos.fpg");
    set_mode(640,480,32);
    write_var(0,60,10,0,alpha_steps);
    idlab = laberinto();
    write_var(0,60,0,0,idlab.alpha);
    Loop
        if(key(_space))while(key(_space))frame; end idlab.alpha=(idlab.alpha-1)%255; end
        if(key(_enter))while(key(_enter))frame; end alpha_steps=(alpha_steps+10)%255;end
        if(key(_esc)) exit(); end
        Frame;
    End
End

Process laberinto()
Begin
        x = 320;
```

```
        y = 240;
        file=fpg;
        graph = 2;
        Loop
            Frame;
        End
End
```

En general, por "flag"(bandera en inglés) se entiende aquel parámetro –o variable según el caso- que dependiendo del valor que tenga, modifica la configuración de algo diferente de una manera o de otra: es como un cajón de sastre donde cada valor que puede tener ese parámetro o variable "flag" cambia un aspecto diferente (de la imagen del proceso, en este caso).

| | |
|---|---|
| ***FLAGS*** | El valor de esta variable activa operaciones especiales que se realizarán en el momento de dibujar el gráfico del proceso (dado por ***GRAPH***) en la pantalla. |

Su valor por defecto es 0, y puede contener uno o la suma de varios de los valores siguientes (si es la suma, los efectos se añadirán uno sobre otro):

| | |
|---|---|
| **1** | Espejado horizontal. El espejado es mucho más rápido de dibujar que una rotación convencional. |
| **2** | Espejado vertical. El espejado es mucho más rápido de dibujar que una rotación convencional. |
| **4** | Transparencia del 50%. Establecer este flag hace que el valor de la variable ***ALPHA*** no se tenga en cuenta: siempre se verá una transparencia del 50% (a añadir a la que ya tenga la propia imagen de 32 bits). |
| **8** | La transparencia se tomará del valor de ***ALPHA*** |
| **16** | Blit aditivo. El blit aditivo (o sustractivo) consiste en hacer que el gráfico sume (o reste) sus componentes de color píxel a píxel con el valor que tenía ese píxel de pantalla antes de dibujar dicho gráfico. Por tanto colores con componentes de color elevadas tienden a la saturación (se acercan al blanco) en el caso del blit aditivo y tienden a 0 en el caso del sustractivo. |
| **32** | Blit sustractivo |
| **128** | Modo "NO_COLOR_KEY" (sin transparencias). Las zonas transparentes de un gráfico fuerzan a hacer transparentes también las zonas coincidentes de los demás gráficos existentes. Para gráficos grandes que no deban respetar transparencias, es interesante utilizar como valor de flags el de NO_COLOR_KEY ya que el dibujado sin transparencia es mucho más rápido que el dibujado con transparencias. |

También existe la posibilidad de asignar como valor de ***FLAGS*** una constante (que equivale a alguno de los valores numéricos anteriores), en vez de directamente dicho valor. Así se gana más claridad en el código. La tabla de correspondencias es la siguiente:

| Valor numérico real | Constante equivalente |
|---|---|
| 1 | ***B_HMIRROR*** |
| 2 | ***B_VMIRROR*** |
| 4 | ***B_TRANSLUCENT*** |
| 8 | ***B_ALPHA*** |
| 16 | ***B_ABLEND*** |
| 32 | ***B_SBLEND*** |
| 128 | ***B_NOCOLORKEY*** |

Comentar también la existencia de otra variable global predefinida diferente, ***TEXT_FLAGS***, cuya finalidad y posibles valores es similar a ***FLAGS***, pero que tan sólo afecta a textos escritos por **write()** o similares.

Un ejemplo trivial de ***FLAGS***:

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";
import "mod_video";

Global
        int fpg;
End

process main()
Private
        int idlab;
End
Begin
    fpg=load_fpg("graficos.fpg");
    set_mode(640,480,32);
    idlab = laberinto();
    write_var(0,60,0,0,idlab.flags);
    Loop
         if(key(_enter)) while(key(_enter))frame; end idlab.flags=(idlab.flags+1)%128;end
         if(key(_esc)) exit();end
         frame;
    End
End

Process laberinto()
Begin
        x = 320;
        y = 240;
        file=fpg;
        graph = 2;
        Loop
            Frame;
        End
End
```

A continuación presento un ejemplo donde se muestra el efecto de establecer la variable ***FLAGS*** a 4 (transparencia). En pantalla se mostrarán tres gráficos superpuestos (un cuadrado rojo, uno verde y otro azul). Como todos tienen transparencia del 50%, se podrán observar las mezclas de colores que se producen en sus interesecciones. Además, uno de los cuadrados aparecerá y desaparecerá de la pantalla a intérvalos regulares, para poder apreciar mejor dichas mezclas.

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";
import "mod_video";

process main()
private
          int grafcuad1, grafcuad2,grafcuad3;
end
Begin
set_mode(640,480,32);
//Genero los gráficos de los tres cuadrados
grafcuad1=new_map(250,250,32);map_clear(0,grafcuad1,rgb(255,0,0));
grafcuad2=new_map(250,250,32);map_clear(0,grafcuad2,rgb(0,255,0));
grafcuad3=new_map(250,250,32);map_clear(0,grafcuad3,rgb(0,0,255));

cuadrado(250,320,grafcuad1);
cuadrado(350,280,grafcuad2);
parpadeo(grafcuad3);
End

Process cuadrado(x,y,graph)
Begin
flags=4;
```

```
Loop
    Frame;
End
End

Process parpadeo(grafcuad3)
Private
     int contador;
end
Begin
cuadrado(150,180,grafcuad3);
Loop
//Ahora que el cuadrado se muestre y desaparezca a intérvalos regulares
      contador++;
      If(contador>10)
           signal(son,s_sleep);
           If(contador>20)
                     signal(son,s_wakeup);
                     Frame(500);
                contador=0;
           End
      End
      If(key(_esc)) exit();End
      Frame;
End
End
```

Otro ejemplo muy curioso donde se utiliza ***FLAGS*** (y ***SIZE***) para mostrar un efecto de cola transparente tras el movimiento de un proceso (a modo de "cometa") es el siguiente:

```
import "mod_key";
import "mod_map";
import "mod_proc";
import "mod_draw";
import "mod_video";

process main()
begin
  set_mode(640,480,32);
  proceso1();
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

PROCESS proceso1()
BEGIN
           graph=new_map(50,50,32);map_clear(0,graph,rgb(130,230,60));
           x=160;y=120;
           LOOP
             if(key(_left)) x=x-10; fantasma(graph,x,y); end
             if(key(_right)) x=x+10; fantasma(graph,x,y); end
             if(key(_up)) y=y-10; fantasma(graph,x,y); end
             if(key(_down)) y=y+10; fantasma(graph,x,y); end
             FRAME;
           END
END

PROCESS fantasma(graph, x, y)
BEGIN
           Z = 1;
           Flags = 4;
           FROM size = 150 to 0 step -10;
                     FRAME;
           END
END
```

Es posible que no quede muy claro el significado del valor 128 de ***FLAGS***. Veremos su significado a partir del siguiente ejemplo, en el cual, en vez de utilizar esta variable local predefinida, utilizaremos la función **map_xput()**, la cual ya sabemos que como último parámetro utiliza los mismos posibles valores de ***FLAGS*** para modificar la imagen que se superpondrá a la imagen-destino.

```
Import "mod_map";
Import "mod_key";
Import "mod_video";

Process main()
private
    int id_map;
    int id_new_map;
    int anchura;
    int altura;
end
Begin
        set_mode(100,100,32);
        id_map = load_png("contrans.png");
        anchura= map_info(0,id_map,G_WIDTH);
        altura= map_info(0,id_map,G_HEIGHT);

        id_new_map = new_map(anchura,altura,32);
        map_clear(0,id_new_map,rgb(100,100,100));
        map_xput(0,id_new_map,id_map,anchura/2,altura/2, 0, 100, 0);
        save_png(0,id_new_map,"sintrans.png");
        unload_map(0,id_new_map);

        id_new_map = new_map(anchura,altura,32);
        map_clear(0,id_new_map,rgb(100,100,100));
        map_xput(0,id_new_map,id_map,anchura/2,altura/2, 0, 100, b_nocolorkey);
        save_png(0,id_new_map,"sintrans2.png");
        unload_map(0,id_map);
End
```

Como se puede observar, lo primero que se hace en el código anterior es cargar una imagen que contendrá transparencias ("contrans.png"), y guardar su ancho y alto. Seguidamente, se crea dinámicamente otra imagen del mismo tamaño que la anterior, toda de un color grisáceo, y a continuación se incrusta con **map_xput()** -valiendo su último parámetro 0- la imagen con transparencias encima de la de color gris, y se graba el resultado en el fichero "sintrans.png". Después se vuelve a repetir exactamente el mismo proceso, pero esta vez valiendo el último parámetro de **map_xput()** 128 (es decir,"b_nocolorkey"), y grabando el resultado en "sintrans2.png". Lo que se obtiene es que con el parámetro "flags" de **map_xput()** igual a 0 todas las zonas transparentes de la imagen "a incrustar" se mantienen transparentes tras la incrustación, permitiendo así poder visualizar el color gris de la imagen que había "por debajo". En cambio, con el parámetro "flags" de **map_xput()** igual a 128, los pixels transparentes de la imagen "a incrustar" fuerzan a que lo sean también los píxeles que coinciden por debajo en la imagen que había "por debajo", con lo que todas las zonas transparentes de la imagen "a incrustar" pasan a ser las zonas transparentes (sin gris) de la imagen final.

Otro ejemplo similar al anterior donde se puede observar el mismo comportamiento es el siguiente, esta vez sí, utilizando la variable ***FLAGS***. En este código existe un proceso con un gráfico con transparencias ("contrans.png") y un fondo de color gris. En el caso de pulsa "Enter", el valor de ***FLAGS*** de ese proceso pasa a ser 128, con lo que los píxeles trasparentes de su gráfico transforman los píxeles de las imágenes que hay por "debajo" suyo en transparentes también, con lo que se puede observar que aparecen zonas negras, ya que al hacer todos los píxeles transparentes, se ve el "fondo pelado" de la pantalla, negro. En cambio, si se pulsa "Space", ***FLAGS*** vale 0, con lo que los píxeles transparentes de la imagen del proceso dejan visualizar los píxeles que hay por debajo, en este caso, los píxeles grises de la imagen de fondo.

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";
import "mod_video";
import "mod_screen";
```

```
process main()
Private
        int idlab;
          int fondo;
End
Begin
    set_mode(640,480,32);
    fondo=new_map(640,480,32);map_clear(0,fondo,rgb(100,100,100));
    put_screen(0,fondo);
    idlab = laberinto();
    write_var(0,60,0,0,idlab.flags);
    Loop
         if(key(_enter)) idlab.flags=128;end
         if(key(_space)) idlab.flags=0;end
         if(key(_esc)) exit();end
         frame;
    End
End

Process laberinto()
Begin
        x = 320;
        y = 240;
        graph=load_png("contrans.png");
        Loop
            Frame;
        End
End
```

Finalmente, indicar que si quisiéramos por ejemplo, un espejado horizontal y vertical a la vez con transparencia, ***FLAGS*** tendría que valer la suma de los diferentes valores que permiten cada efecto individual, es decir: 1+2+4=7.

## *Tutorial de un ping-pong

En este apartado vamos a crear nuestro segundo juego escrito en Bennu: el ping-pong clásico de toda la vida: con dos palas a los lados que se moverán arriba y abajo y una pelota que irá rebotando en ellas o perdiéndose fuera de la pantalla si algún jugador no es suficientemente rápido.

Lo primero, como siempre, los gráficos. Sólo necesitaremos dos: el de la pelota y el de las dos palas. El gráfico de las dos palas será un rectángulo blanco de 30 píxeles de anchura x 100 píxeles de alto, y el gráfico de la pelota será un círculo blanco de unos 30 píxeles de diámetro. Ambas imágenes las incluiremos en un archivo FPG llamado "pingpong.fpg" (el rectángulo que representa las palas con código 001 y el círculo que representa la pelota con código 002).

Comencemos. Lo primero que vamos a escribir va a ser el esqueleto semivacío de nuestro programa, incluyendo la creación desde el programa principal de los diferentes procesos necesarios, de tal manera que empecemos visualizando todos los elementos que van a jugar un papel en nuestro programa (bola y palas). Es posible que, en este momento, te preguntes si es necesario que creemos un proceso diferente para cada pala (izquierda y derecha), o si con un mismo proceso ya vale, ya que tienen el mismo gráfico y su comportamiento es muy similar.(igual que hemos hecho en el juego del laberinto con los enemigos). Si optáramos por esta última posibilidad, sólo habría que indicar unas coordenadas en pantalla diferentes para cada pala, pero serían el mismo proceso. No obstante, la respuesta es clara: necesitarás un proceso por cada pala. Las razón más sencilla y rápida de entender es que para mover cada pala independientemente una de la otra, necesitarás definir teclas diferentes de movimiento. Es decir, si queremos que la pala izquierda se mueva arriba podríamos hacer que esto ocurra pulsando la tecla "q", pero si queremos que sea la pala derecha la que suba, tendremos que utilizar otra tecla diferente. Si tuviéramos un sólo proceso, pulsando "q" subirían las dos palas a la vez..Sólo por eso, ya tenemos un motivo para hacer dos procesos separados.

Así pues, el código que será nuestro punto de partida será algo así como éste (espero que no tengas

ningún problema en entenderlo):

```
import "mod_map";
import "mod_key";
import "mod_video";

global
          int id1;
end

process main()
begin
          set_mode(640,480,32);
          id1=load_fpg("pingpong.fpg");
          pala1();
          pala2();
          bola();
end


process pala1()
begin
          x=90;
          y=240;
          file=id1;
          graph=1;
          loop
                    if(key(_q)) y=y-10;end
                    if(key(_a)) y=y+10;end
                    frame;
          end
end

process pala2()
begin
          x=550;
          y=240;
          file=id1;
          graph=1;
          loop
                    if(key(_up)) y=y-10;end
                    if(key(_down)) y=y+10;end
                    frame;
          end
end

process bola()
begin
          x=320;
          y=240;
          file=id1;
          graph=2;
          loop
                    frame;
          end
end
```

Vemos que el programa principal se encarga básicamente de crear los tres procesos involucrados en el juego, y ¡atención! morir seguidamente. Es casi la primera vez que no vemos un bucle LOOP/END en un programa principal. Te preguntarás por qué no se acaba entonces la ejecución: bueno, porque aunque el código principal ya no esté en ejecución, él antes ha puesto en marcha tres procesos (“pala1”,”pala2”, y “bola”) los cuales sí tienen un bucle LOOP/END, con lo que estos tres procesos estarán ejecutándose en paralelo perpétuamente. Mientras quede un sólo proceso en funcionamiento, el programa todavía estará corriendo.

Fíjate también que ya hemos hecho que las dos palas se muevan para arriba y para abajo con sendas teclas. No hemos limitado su movimiento vertical a fuera de la pantalla, pero sería fácil hacerlo.

Ahora lo que nos falta es hacer que la bola se mueva, y que al rebotar en alguna de las palas, cambie su dirección de movimiento. Además haremos que si la bola sale de los límites de la pantalla (esto es, ninguna pala ha llegado a tocarla), se acabe el programa. Para ello, vamos a implementar un sistema provisional que luego mejoraremos. La idea es que nada más crearse el proceso "bola()", ésta empiece a moverse mediante la función **advance()**. Como esta función dirige el gráfico en el sentido que viene marcado por su variable ***ANGLE***, y como el valor de esta variable al inicio de cada proceso es 0 por defecto, la bola empezará a moverse hacia la derecha horizontalmente. Así pues, lo primero que haremos es que cuando la bola colisione con la pala derecha (la primera que se encontrará), la bola cambie su orientación y se siga moviendo en esa nueva orientación.

El cálculo de la nueva orientación se puede hacer de muchas maneras más o menos complicadas, pero ahora lo haremos fácil: ya que la bola inicialmente se mueve horizontalmente lo que haremos de momento será que sólamente se pueda mover en esta dirección, pero cambiando el sentido. O sea, si la bola va de izquierda a derecha antes de impactar con la pala, después tendrá que ir de derecha a izquierda. Para ello, fíjate lo que tenemos que hacer: la bola inicialmente va a la derecha porque su ***ANGLE*** vale 0. Si quisiéramos que fuera hacia la izquierda, tendría que valer 180 grados. Así que lo que haremos será que cuando colisione con la "pala1()", *ANGLE* valga 180000. Para ello, modificamos el proceso "bola()" de la siguiente manera (el resto del código permanece igual):

```
process bola()
begin
          x=320;
          y=240;
          file=id1;
          graph=2;
          loop
                    advance(10);
                    if(collision(type pala2))angle=180000;end
                    frame;
          end
end
```

Si lo pruebas (importa en el código los módulos pertinentes), verás que efectivamente, al chocar la bola con la pala de la derecha, rebota y va directa a la pala de la izquierda, la cual no reacciona porque todavía no lo hemos dicho que lo haga. Es lo siguiente que haremos.

Un paréntesis antes de continuar: es posible que te hayas preguntado dónde colocar la detección de la colisión: o en el proceso "bola()" o en el proceso "pala2()". Si hubiéramos optado por hacerlo en "pala2()" no habría sido más difícil que escribir (no lo hagas) la siguiente condición dentro de su código:

```
if(collision(type bola)) identificadorProcesoBola.angle=180000;end
```

No obstante, y esto es una recomendación para todos los juegos que hagas, es MUCHO MAS fácil, cómodo y limpio detectar las colisiones en aquel proceso que vaya a colisionar con más procesos. Es decir, "bola()" colisionará con "pala1()" y "pala2()". En cambio, las palas sólo colisionarán con "bola()". Esto quiere decir que si detectáramos las colisiones en las palas, ya hemos visto que tendríamos que escribir una línea como la anterior en "pala2()" pero otra de muy similar en el código de "pala1()", con lo que tendríamos en dos procesos distintos la colisión con un proceso "bola()". ¿No es mucho más fácil detectar desde el propio proceso "bola()" las posibles colisiones que éste puede tener con los demás procesos -en este caso, "pala1()" y "pala2()"-? Es por tanto buena idea centralizar la detección de colisiones en aquél proceso que vaya a colisionar con más procesos diferentes.

Bueno, volvamos a nuestro ping-pong. ¿Cómo hacer para que cuando la bola choque con la pala de la izquierda ("pala1()"), cambie su sentido otra vez 180 grados para volver a dirigirse horizontalmente hacia la derecha? Viendo lo que hemos escrito antes, parece claro que lo que tendremos que escribir justo antes del frame del loop de "bola()" es una línea como ésta:

```
if(collision(type pala1))angle=0;end
```

¡Ya tenemos nuestra bola que rebota! Ahora lo que nos falta es hacer que cuando la bola se pierda en la inmensidad, fuera de los límites de la pantalla, porque una pala se ha movido y no ha logrado colisionar con ella,el programa se acabe. Esto es simple, sólamente hay que añadir una última línea dentro del loop de "bola()"

```
if(x<0 or x>640) exit(); end
```

Nuestro ping-pong ya funciona tal como en principio hemos decidido. No obstante, vamos a hacer el código fuente un poco "más elegante": sustituye los dos ifs que detectan las colisiones de "bola()" con "pala1()" y "pala2()" dentro del código de "bola" por este único if:

```
if(collision(type pala1) or collision(type pala2))angle=(angle+180000)%360000;end
```

Esta línea hace lo mismo que las otras dos que hemos quitado. ¿Cómo lo hace? Para empezar, la condición comprueba si la bola colisiona con "pala1()" Ó "pala2()". En cualquiera de estas dos colisiones, hará lo mismo, que es cambiar el ***ANGLE*** de la bola por un valor obtenido de una expresión un tanto exótica.Estudiemos qué valores obtenemos: inicialmente, cuando la bola se mueve horizontalmente a la derecha, su ***ANGLE*** es 0, así que cuando choca con "pala2()", su nuevo ***ANGLE*** es el que tenía antes (es decir, 0) más 180000 módulo 360000: es decir, 180000. Correcto. Ahora la bola va horizontalmente hacia la izquierda. Cuando choca con "pala1()" su nuevo ***ANGLE*** es el que tenía antes (es decir, 180000) más 180000 módulo 360000, es decir 360000 módulo 360000, es decir, 0. ¡Volvemos a la situación inicial, con la bola dirigida horizontalmente a la derecha! Es decir, que con una expresión un poco más ingeniosa, podemos hacer las cosas más bien hechas.

No obstante, este ping-pong todavía tiene mucho que mejorar: la pelota sólo se mueve en horizontal, no hay marcador de puntuaciones, no se vuelve a empezar la partida cuando la bola sale fuera de la pantalla...Vamos a intentar solventar estas carencias. Lo único que será modificado a partir de ahora respecto el código que tenemos en estos momentos es el proceso "bola()": el resto de procesos y el código principal serán los mismos.

Un consejo previo: por cada cosa que cambiemos en el código, pruébalo a ejecutar, para ir observando cómo el juego va cambiando poco a poco: si introduces todos los cambios de golpe y lo pruebas al final, no habrás notado las pequeñas mejoras que pasito a pasito se han ido incorporando, y te habrás perdido parte del encanto de este tutorial.

Bien. Lo primero que vamos a hacer es que la bola se pueda mover en cualquier dirección de la pantalla, no solamente en horizontal. Para ello, haremos que cuando el proceso "bola()" se cree, su orientación sea aleatoria. Esto consiste simplemente en añadir justo antes (fuera) de su bucle Loop la línea

```
angle=rand(0,360000);
```

Y también cambiaremos lo de que si la bola excede los límites laterales de la pantalla el juego se acaba. Ahora haremos que cuando ocurra esto, la bola se resitúe en el centro de la pantalla, y además, con una nueva orientación aleatoria, como si volviéramos a empezar. Para hacer esto, has de quitar el if que ejecutaba la orden **exit()** y poner en su lugar estos dos ifs:

```
if(x<0)
            x=320;
            y=240;
            angle=rand(0,360000);
end

if(x>640)
            x=320;
            y=240;
            angle=rand(0,360000);
end
```

¿Por qué escribimos dos ifs diferentes si hacen lo mismo? Con una condición OR -como estaba antes- ya valdría...Sí, es cierto, pero lo hemos hecho así pensando en los puntos de cada jugador. No es lo mismo que la bola haya salido por el lado izquierdo de la pantalla (momento en el que el jugador que maneja la pala derecha aumentará un punto en su marcador) que la bola haya salido por el lado derecho (momento en el que el jugador que maneja la pala izquierda aumentará un punto en su marcador). Dentro de cada uno de los dos ifs tendremos que aumentar en uno la puntuación de un jugador diferente. Es por eso que tenemos que escribir dos. Vamos a ello: a continuación presento el código del proceso "bola()" tal como tiene que quedar (las líneas nuevas están marcadas en negrita) para que se vea la puntuación de cada jugador (importa los módulos pertinentes):

```
process bola()
private
            int punts1,punts2;
```

```
end
begin
        x=320;
        y=240;
        file=id1;
        graph=2;
        angle=rand(0,360000);
        write_var(0,90,50,4,punts1);
        write_var(0,550,50,4,punts2);
        loop
                advance(10);
                if(collision(type pala1) or collision(type pala2))
                        angle=(angle+180000)%360000;
                end
                if(x<0)
                        punts2=punts2+1;
                        x=320;
                        y=240;
                        angle=rand(0,360000);
                end
                if(x>640)
                        punts1=punts1+1;
                        x=320;
                        y=240;
                        angle=rand(0,360000);
                end
                frame;
        end
end
```

Vemos que hemos declarado dos variables privadas, *punts1* y *punts2* que representan respectivamente los puntos del jugador que maneja "pala1()" y del que maneja "pala2()". Vemos también que mostramos los puntos por pantalla mediante **write_var()** (actualizándose cuando sea necesario,pues, sin necesidad de escribir texto dentro del loop -con el engorro de estar usando **delete_text()** para evitar el error de imprimir demasiados textos en pantalla-). Y vemos que si la bola sale por la izquierda de la pantalla, aumenta un punto en el marcador del jugador de la pala 2, y si sale por la izquierda, aumenta un punto en el marcador del jugador de la pala 1.

Ya que estamos, podríamos hacer ahora que cuando se llegara a un máximo de puntuación, el juego acabara. Por ejemplo, si un jugador cualquiera llega a 3 puntos, ya ha ganado y se acaba el juego.¿Cómo hacemos esto? Por ejemplo, escribiendo el siguiente if justo antes de la línea Frame; del Loop del proceso "bola()".

```
if (punts1 == 3 or punts2 == 3)
        while(not key(_enter))
                delete_text(0);
                write(0,320,200,4,"GAME OVER!");
                frame;
        end
        exit();
end
```

Estudiemos este if. Lo que dice es que si cualquiera de los dos jugadores llega a tres puntos (es decir, si *punts1* o *punts2* vale 3), nos meteremos en un bucle, del cual sólo podremos salir si pulsamos la tecla Enter. Cuando la pulsemos, se acabará el programa. Y mientras no la pulsemos, lo que veremos será un simple texto centrado en la pantalla que dice "Game Over". Fíjate que necesitamos poner la orden **delete_text()** para evitar que el bucle while imprima demasiados "Game Over" uno encima de otro y nos dé el error ya conocido de tener demasiados textos en pantalla. También necesitamos poner una orden Frame; porque recuerda que para que se vea ese texto (y el programa no se quede colgado), necesitamos lanzar un fotograma.

Fíjate también que cuando hemos llegado al Game Over y no hemos pulsado todavía el Enter (y por tanto, estamos dentro del while), podemos seguir moviendo las palas tranquilamente -son procesos independientes- , y la bola se ha vuelto a situar en el centro de la pantalla con una orientación al azar, pero no se mueve. ¿Por qué no se mueve? Porque para hacerlo se ha de ejecutar la línea que consta de la orden advance(), al principio del Loop de este proceso, cosa que ahora es imposible hacer porque estamos encerrados dentro de las iteraciones infinitas del while y no podemos salir de allí, a no ser que pulsemos el Enter, el cual no es solución porque en ese momento saldremos ya del programa.

Y llegamos finalmente a la parte posiblemente más complicada: la gestión de los rebotes. Hay muchas maneras de simularlos, y muchas maneras más óptimas que la que se va a presentar aquí, pero en general, las maneras más realistas y sofisticadas necesitan por parte del programador de unos conocimientos elevados de geometría y de física. Nosotros ahora no implementaremos ningún sistema de choques, deformaciones ni rebotes basado en ninguna ley física ni formularemos grandes ecuaciones, sino que programaremos un sistema sencillamente pasable y mínimamente jugable.

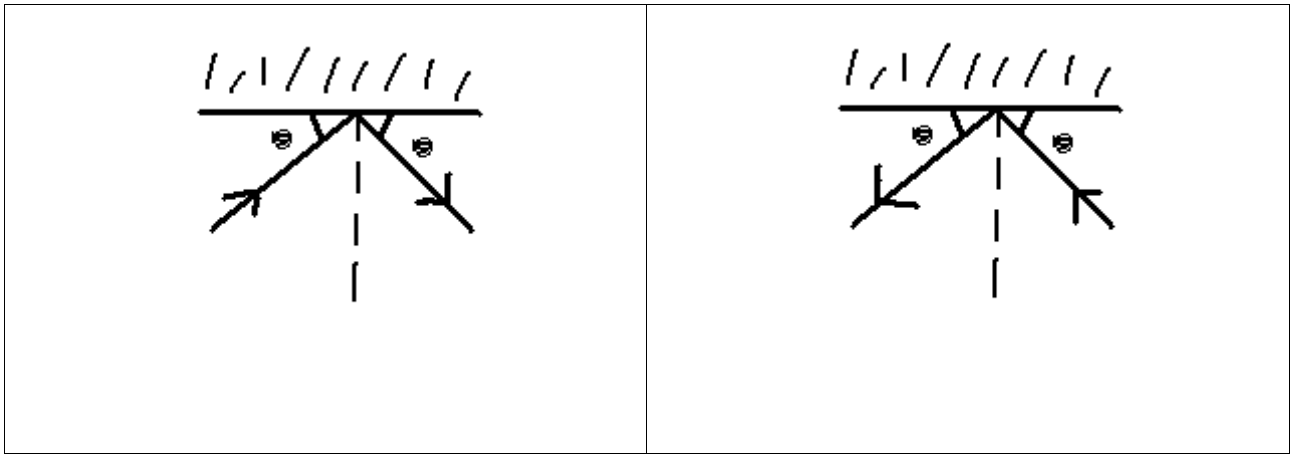
Lo que haremos será lo siguiente: distinguiremos (porque así lo queremos, no por otro motivo) entre los choques de la bola contra los límites inferior y superior de la pantalla y los choques de la bola contra las dos palas:

*En los choques que sufre la bola contra los extremos inferior y superior de la pantalla, el ángulo de salida de la bola respecto la superficie de choque será el mismo que el ángulo de incidencia de la bola respecto ésta. O dicho de otra manera, el ángulo de reflexión será el mismo que el de incidencia (respecto la superficie, aunque de hecho, respecto la perpendicular también sería igual).

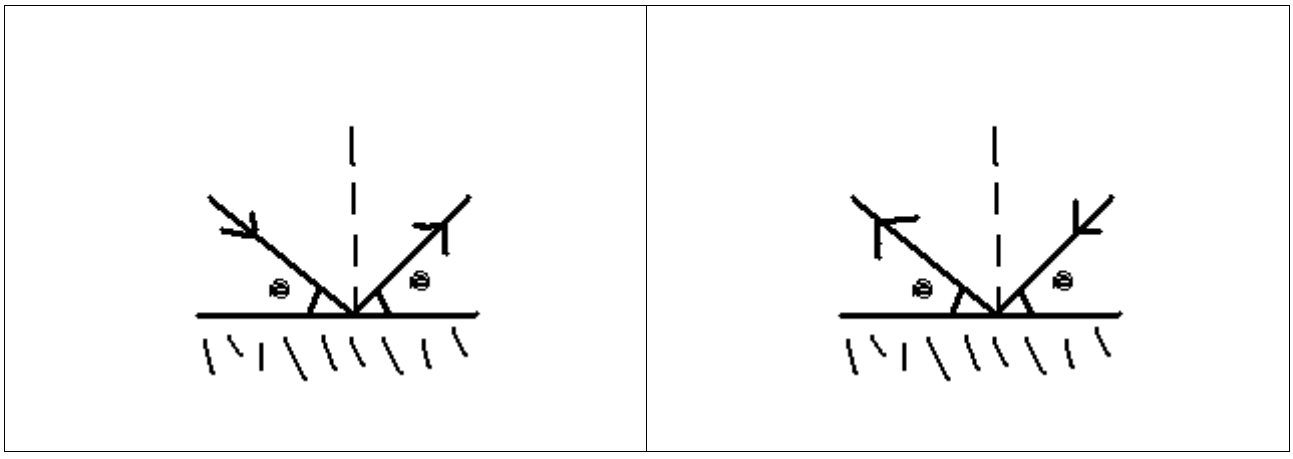
*En los choques que sufre la bola contra las palas, para hacer el juego más interesante e imprevisto, lo que haremos será que el ángulo de salida de la bola respecto la superficie de choque sea un ángulo aleatorio, pero siempre dentro del mismo cuadrante donde estaría el ángulo de reflexión si éste fuera el mismo que el de incidencia (el caso anterior).

Una vez claro este punto, debemos estudiar los posibles casos con que podemos encontrarnos, que son varios.

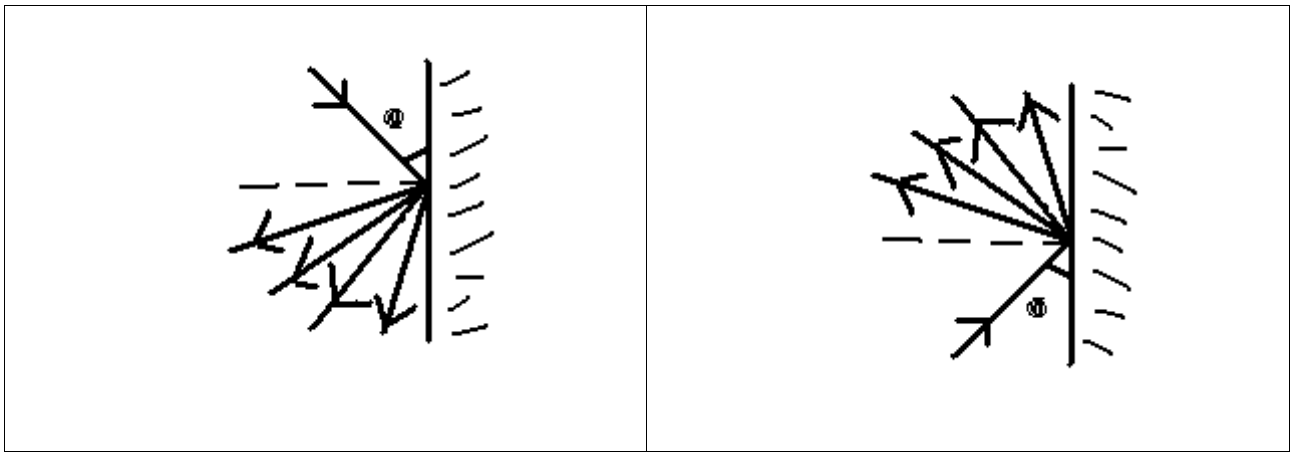
*Si la bola choca contra el límite superior de la pantalla, puede hacerlo viniendo desde la izquierda (hacia la derecha) o viniendo desde la derecha (hacia la izquierda), o sea:



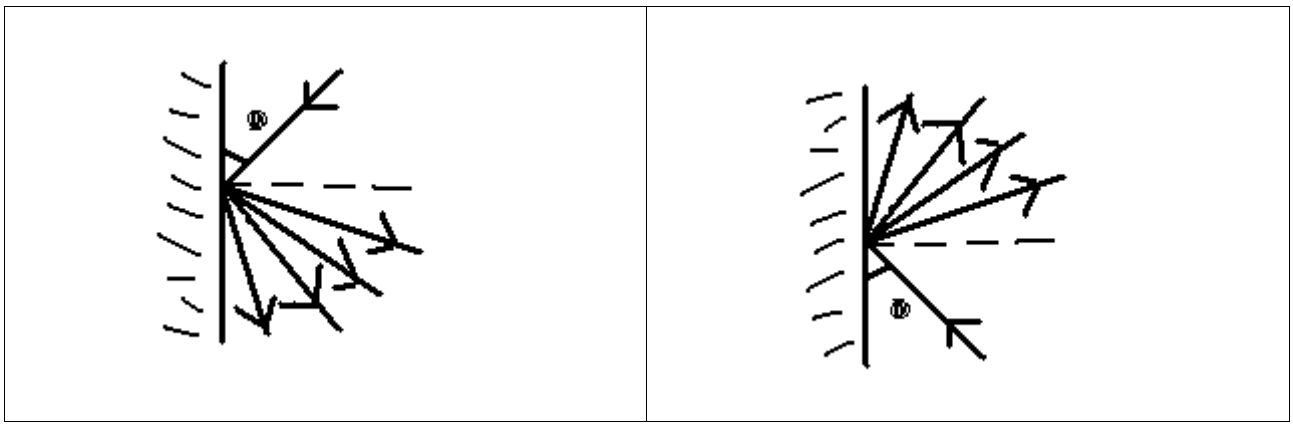
*Si la bola choca contra el límite inferior de la pantalla,puede hacerlo viniendo desde la izquierda (hacia la derecha) o viniendo desde la derecha (hacia la izquierda), o sea:



*Si la bola choca contra la pala derecha ("pala2()"), puede hacerlo viniendo desde arriba (hacia abajo) o viniendo desde abajo (hacia arriba), o sea:



*Si la bola choca contra la pala izquierda ("pala1()"), puede hacerlo viniendo desde arriba (hacia abajo) o viniendo desde abajo (hacia arriba), o sea:



A partir de aquí, deberíamos coger lápiz y papel y empezar a dibujar triangulitos rectángulos para descubrir equivalencias de ángulos gracias a que sabemos de la escuela que la suma de los tres ángulos de cualquier triángulo es 180º. Si hacemos esto con calma y además tenemos en cuenta que ***ANGLE*** siempre será el ángulo entre la línea horizontal y donde apunte el gráfico del proceso, nos saldrán las siguientes expresiones para conseguir que la orientación de la bola después del choque sea la que nosotros queremos:

*Si la bola choca contra el límite superior viniendo desde la izquierda (es decir, con ***ANGLE***>0º y <=90º):

*angle=(360000-angle)%360000;* o lo que es lo mismo: *angle=-angle;*

*Si la bola choca contra el límite superior viniendo desde la derecha (es decir, con ***ANGLE***>90º y <=180º):

*angle=(360000-angle)%360000;* o lo que es lo mismo: *angle=-angle;*

*Si la bola choca contra el límite inferior viniendo desde la izquierda(es decir,con ***ANGLE***>180º y <=270º):

*angle=(360000-angle)%360000;* o lo que es lo mismo: *angle=-angle;*

*Si la bola choca contra el límite inferior viniendo desde la derecha (es decir, con ***ANGLE***>270º y <=360º):

*angle=(360000-angle)%360000;* o lo que es lo mismo: *angle=-angle;*

*Si la bola choca contra la pala derecha viniendo desde la arriba (es decir, con ***ANGLE***>270º y <=360º):

*El nuevo **ANGLE** puede tener cualquier valor entre 180º y 270º*

*Si la bola choca contra la pala derecha viniendo desde la abajo (es decir, con ***ANGLE***>0 y <=90º):

*El nuevo **ANGLE** puede tener cualquier valor entre 90º y 180º*

*Si la bola choca contra la pala izquierda viniendo desde la arriba (es decir, con ***ANGLE***>180º y <=270º):

*El nuevo **ANGLE** puede tener cualquier valor entre 270º y 360º*

*Si la bola choca contra la pala izquierda viniendo desde la abajo (es decir, con ***ANGLE***>90º y <=180º):

*El nuevo **ANGLE** puede tener cualquier valor entre 0º y 90º*

Sorprendentemente, en todos los casos donde chocamos contra los límites inferior o superior de la pantalla nos ha salido la misma expresión, y vemos también que las expresiones son diferentes para las dos palas, incluso dependiendo de la orientación inicial de la bola en cada una de ellas.

Una vez que ya hemos encontrado las expresiones adecuadas a cada caso, lo único que nos falta es escribirlas convenientemente en nuestro código fuente, y ya habremos acabado nuestro juego. A continuación presento el código fuente completo del proceso "bola()", y en negrita las líneas que tenemos que añadir para implementar todo lo que acabamos de discutir sobre los choques contra los límites de la pantalla y las palas

```
process bola()
private
          int punts1,punts2;
end
begin
          x=320;
          y=240;
          file=id1;
          graph=2;
          angle=rand(0,360000);
          write_var(0,90,50,4,punts1);
          write_var(0,550,50,4,punts2);
          loop
                    advance(15);

/*Hemos quitado la sentencia que teníamos hasta ahora que controlaba el chocque de la
bola con las dos palas, y hemos separado en dos ifs las colisiones dependiendo de si la
bola choca con pala1 o pala2, ya que su orientación será diferente dependiendo de con
quien choque*/
                    if(collision(type pala1))//Pala izquierda
                              //La bola viene de abajo
                              if(angle > 90000 and angle < 180000)
                                        angle=rand(0,90000);//Va al primer cuadrante
                              end
                              //La bola viene de arriba
                              if(angle > 180000and angle < 270000)
                                        angle=rand(270000,360000);//Va al 4º cuadrante
                              end
                    end

                    if(collision(type pala2))//Pala derecha
                              //La bola viene de arriba
                              if(angle > 270000 and angle < 360000)
                                        angle=rand(180000,270000); //Va al 3r cuadrante
                              end
                              //La bola viene de abajo
                              if(angle > 0 and angle < 90000)
                                        angle=rand(90000,180000); //Va al 2º cuadrante
                              end
                    end
```

```
/*Si la bola choca con el límite superior o inferior de la pantalla, venga de donde
venga, se comportará igual*/
                    if(y<0 or y>480)
                              angle=(360000-angle)%360000;
                    end

                    if(x<0)
                              punts2=punts2+1;
                              x=320;
                              y=240;
                              angle=rand(0,360000);
                    end
                    if(x>640)
                              punts1=punts1+1;
                              x=320;
                              y=240;
                              angle=rand(0,360000);
                    end

                    if (punts1 == 3 or punts2 == 3)
                              while(not key(_enter))
                                                  delete_text(0);
                                                  write(0,320,200,4,"GAME OVER!");
                                                  frame;
                              end
                              exit();
                    end
                    frame;
          end
end
```

En verdad, el movimiento de la bola y la gestión de sus choques se podría haber hecho de una manera mucho más simple e igualmente efectiva: sólo con un poco de ingenio, y sin tanto lío con los ángulos. A continuación presento el código completo de otro juego de pingpong muy similar (funcionará con el mismo FPG del ejemplo anterior), donde se puede apreciar una técnica diferente para hacer que la bola se mueva y detecte los choques con la parte superior e inferior de la pantalla y con las dos palas.

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_grproc";
import "mod_math";
import "mod_rand";

global
/*Cuando "fin" valga 1 (es decir, cuando la bola salga por primera vez por la derecha o
la izquierda de la pantalla, el juego acabará*/
          int fin = 0;
end

process main()
begin
          load_fpg("pingpong.fpg");
          bola(160,120);
          pala1(5,120);
          pala2(315,120);
          loop
/*Este if estará pendiente de si el juego ha acabado. Si es así,y se pulsa "x", se crea
otra vez el proceso bola (que está muerto en este momento) y se vuelve a empezar a
jugar*/
                    if (fin == 1 and key(_x))
                      delete_text(0); //Elimino el texto del GameOver
                      bola(160,120);
                      fin = 0;
                    end
                    frame;
```

```
          end
end

process pala1(x,y)
begin
          graph=1;
          loop
                    if(key(_q) and y>10)y=y-5;end
                    if(key(_a) and y<230)y=y+5;end
                    frame;
          end
end

process pala2(x,y)
begin
          graph=1;
          loop
                    if(key(_up) and y>10)y=y-5;end
                    if(key(_down) and y<230)y=y+5;end
                    frame;
          end
end

process bola(x,y)
private
//Cantidad de píxeles que se moverá la bola cada frame en dirección horizontal
          int dx = 5;
//Cantidad de píxeles que se moverá la bola cada frame en dirección vertical
          int dy = 5;
//Complemento a dy para calcular el movimiento vertical de forma más compleja
          int dys=1;
end
begin
          graph = 2;
          loop
//Las dos líneas siguientes definen el movimiento de la bola en cada frame
                    x=x+dx;
                    y=y+dy;
                    if(collision(type pala1) or collision(type pala2))
//Invierto el movimiento horizontal (derecha <-> izquieda)
                              dx = -dx;
/*Las dos lineas siguientes establecen el valor del incremento vertical del movimiento de
la bola. Lo primero que hago es hacer que "dys" valga 1 o -1 dependiendo del valor actual
de "dy" y de su valor absoluto (ABS() es un comando de Bennu que devuelve el valor
absoluto del valor numérico que se le ponga como parámetro, se explicará en el capítulo
8). Si actualmente "dy" es positivo, "dys" valdrá 1. Si actualmente "dy" es negativo,
"dys" valdrá -1. Una vez que "dys" vale 1 o -1, se establece el nuevo incremento en la
posición y de la bola a partir de un número aleatorio entre 1 y 5 multiplicado por 1 ó
-1. Tal como hemos dicho, si el anterior "dy" era positivo, "dys" será 1 y los valores
posibles del nuevo "dy" serán 1,2,3,4 o 5, aleatoriamente. Si el anterior "dy" era
negativo, "dys" valdrá -1 y los valores posibles del nuevo "dy" serán -1,-2,-3,-4 o -5,
aleatoriamente. Fíjate que si la bola baja ("dy" positivo), continúa bajando ya que "dy"
continúa siendo positivo- aunque posiblemente con otro valor, y por tanto, a otra
velocidad-, y si la bola sube ("dy" negativo), continúa subiendo.Sólo se cambiará el
sentido vertical cuando la bola choque contra los límites inferior o superior de la
pantalla, no contra las palas.*/
                              dys = dy/abs(dy);
                              dy = rand(1,5)*dys;
                    end
/*Si la bola choca contra los límites inferior o superior de la pantalla, invierto el
movimiento vertical (arriba <-> abajo)*/
                    if(y<3 or y>237)
                              dy = -dy;
                    end
/*Si la bola sale fuera de los límites izquierdo o derecho de la pantalla, llamo al
proceso "gameover" y hago un suicidio del proceso "bola" mediante return*/
                    if(x<0 or x>320)
                              gameover();
```

```
                                return;
                        end
                        frame;
                end
end;

/*Proceso que simplemente escribe un texto en pantalla y asigna a "fin" el valor de 1,
con lo que el if del programa principal ya se podrá ejecutar si se pulsa "X", para volver
a jugar.Este proceso es muy pequeño y se podría haber eliminado, escribiendo su código
directamente en el mismo lugar donde se le llama*/
process gameover()
begin
        write(0,160,110,4,"GameOver");
        write(0,160,120,4,"Pulsa X para volver a empezar");
        fin = 1;
end
```

A continuación presento un nuevo código de un juego de ping-pong (¡otro más!) donde podemos encontrar otra manera mucho más fácil todavía de gestionar los choques de la bola contra las palas (con una única fórmula simple: ¡averigua con lápiz y papel cómo se llega ella!) , y además tenemos el detalle interesante de que las dos palas están controladas por el mismo proceso "pala()", en vez de con dos procesos independientes. Como novedad, la velocidad de la bola va aumentando a medida que se juega. Para poderlo ejecutar necesitarás el mismo FPG de los ejemplos anteriores, "pingpong.fpg".

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_grproc";
import "mod_math";
import "mod_rand";
import "mod_proc";

global
/*El elemento score[0] es la puntuación del jugador de la izquierda y el elemento
score[1] es la puntuación del jugador de la derecha*/
        int score[1];
        int fpg;
end

begin
        fpg=load_fpg("pingpong.fpg");
        bola();
        pala(20,_up,_down);
        pala(300,_q,_a);

        write(0,160,20,4,"-");
        write_var(0,100,20,4,score[0]);
        write_var(0,220,20,4,score[1]);

        loop
                if(key(_esc)) exit(); end
                frame;
        end
end


Process bola()
private
        int speedo; //Velocidad de la bola
/*Guarda el código identificador del proceso pala en caso de que colisione con él*/
        int pala_id;
end
begin
  graph=2;
        x = 160;
        y = 120;
        angle = rand(0,360) * 1000;
```

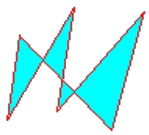
```
          speedo = 500; //Velocidad inicial. Irá aumentando progresivamente
          loop
/*Si la bola está a punto de superar el límite inferior o superior de la pantalla (si la
bola tiene un radio de 5 píxeles...para que no se deje de ver entera*/
          if(y<5 or y>235)
              //El ángulo de entrada es el ángulo de salida!
                        angle = -angle;
          end
/*Si la bola sale por la izquierda, punto para el jugador con score[0]*/
          if(x<0)
                        score[0]=score[0]+1;
                        //Se resetea la velocidad a la estandar

                        speedo = 500;
                         //Se resetea la posición de la bola
                        y = 120;
                        x = 220;
          end
/*Lo mismo para el extremo derecho de la pantalla, sólo que ahora con el marcador de la
otra pala*/
          if(x>320)
                        score[1]++;
                        speedo = 500;
                        y = 120;
                        x = 100;
          end
          //Si choca contra una pala, SEA CUAL SEA
          if(pala_id = collision(type pala))
/*Con esta fórmula tan sencilla obtenemos la nueva orientación de la bola después del
choque*/
                       angle = -angle + 180000;
                        advance(3);
            end
           //Se incrementa lentamente la velocidad a cada frame
           speedo=speedo+3;
           //Mueve la bola en la dirección actual
           advance(speedo/100);
          frame;
   end //loop
end


/*Creo un único proceso "pala" para las dos palas, pasando como parámetro las teclas que
regirán el movimiento de cada una de ellas, y su posición horizontal (que son las únicas
cosas que diferencian una pala de la otra, porque hemos visto que con una misma fórmula
se pueden gestionar los choques de la bola con las dos palas.*/
Process pala(x,int keyup,int keydown)
begin
          y = 120;
          graph=1;
          loop
                    if(key(keyup)) y =y-12; end
                        if(key(keydown))y =y+12; end
/*Limito los movimientos de las raquetas a los extremos de la pantalla*/
                    if(y < 20) y=20; end
                        if(y > 220)y=220; end
                       frame;
          end
end
```

## *Variante del ping-pong: el juego del "Picaladrillos"

A partir del código básico del ping-pong podemos crear múltiples juegos sencillos que se basan en los mismos principios de colisiones con bolas y rebotes. Todo dependerá de hasta dónde llega nuestra imaginación. Como

ejemplo, presento a continuación un juego muy sencillo, el “Picaladrillos”. El jugador controla una raqueta que puede mover solamente de forma horizontal con los cursores, la cual dirigirá una pelota cuya función es destruir (colisionando con ellos) una serie de ladrillos colocados a lo largo del borde superior de la pantalla. Si la bola cae por el borde inferior de la pantalla se pierde: para generar otra bola habrá que apretar la tecla CTRL. El objetivo es destruir todos los ladrillos.

Para probar el juego, necesitarás un fpg llamado “breakout.fpg” compuesto por tres imágenes. La imagen 1 será la bola (de 25x25 está bien, pero no es un tamaño obligatorio), la imagen 2 será un ladrillo (que para este código de ejemplo concreto sí es conveniente -por las medidas de pantalla implicadas- que sus medidas sean 28x138) y la imagen 3 será la pala (con un tamaño de 120x120 está bien, y que para hacerlo más interesante podría tener una forma irregular, como ésta)



```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_grproc";
import "mod_proc";
import "mod_math";

global
     int blocksleft;//Contará la cantidad de ladrillos que hay aún
end

process main()
private
          int i,j;//Variables para los bucles
end
begin
          load_fpg("breakout.fpg");
          paddle(160,220); //Raqueta
          ball(160,80);//Bola
/*Los dos for siguientes sirven para dibujar en pantalla los ladrillos que tendremos que
destruir. La estructura es sencilla: el primer bucle sirve para darle 10 posiciones
horizontales diferentes a los ladrillos, y el segundo sirve para dar 3 posiciones
verticales. Así que cada vez que lleguemos a la llamada al proceso block, dibujaremos un
ladrillo en una de las 10 posiciones horizontales posibles y en una de las 3 posiciones
verticales posibles, con lo que al final tendremos 30 ladrillos distribuidos en 3 filas y
10 columnas*/
          for(i=0;i<10;i++)
                    for(j=0;j<3;j++)
//Como creamos un ladrillo,tenemos que añadirlo al contador
                              blocksleft++;
/*Dibujamos el ladrillo propiamente dicho.Los números están para ajustar la posición a
las medidas de la pantalla.Recuerda que para que cuadre el gráfico del ladrillo ha de
tener  unas dimensiones de 28x138*/
                              block(i*32+16,j*16+8);
                    end
          end
          write(0,180,230,5,"Ladrillos que faltan: ");
          write_var(0,180,230,3,blocksleft);
          loop
//Puedes crear una bola extra cuando quieras
                    if(key(_control))
//Esperar hasta que la tecla se suelte
                                   while(key(_control))frame;end
                                   ball(160,80);
                     end
                     frame;
```

```
          end
end

process block(x,y)
begin;
          graph=2;
          loop
                    frame;
          end
end

process paddle(x,y)
begin
          graph=3;
          loop
                    if(key(_left))x=x-5;end
                    if(key(_right))x=x+5;end
                    if(x<15)x=15;end
                    if(x>305)x=305;end
                    frame;
          end
end

process ball(x,y)
private
          int vx=0;//Su velocidad inicial (cambiará) en la dirección X e Y
          int vy=5;
          int paddle;
          int block;
end
begin
          graph=1;
          loop
//Para no tener que escribir tanto después en los ifs
                    paddle=COLLISION(type paddle);
                    block=COLLISION(type block); //Lo mismo
//Si la bola choca contra un ladrillo
                    if(block!=0)
//Comprobamos que hemos chocado contra la parte inferior o superior de un ladrillo...
                              if(abs(block.y-y+vy)>8)
                                        vy=-vy;//...e invertimos la velocidad vertical
                              end
//Comprobamos que hemos chocado contra la parte izquierda o derecha de un ladrillo...
                              if(abs(block.x-x+vx)>16)
                                        vx=-vx;//...e invertimos la velocidad horizontal
                              end
/*La bola no nesariamente choca sólo contra un ladrillo a la vez, sobretodo cuando incide
de forma horizontal sobre ladrillos superpuestos, donde choca con todos ellos a la vez.*/
                              while(block!=0)
                                        blocksleft--;
                                        signal(block,s_kill);
/*Vemos si hay otro ladrillo con el que haya colisionado.(Recordad que cada vez que se
llama a "collision", se da el siguiente item con el que se ha colisionado, hasta que se
haga un frame, momento en el que se "resetea" la lista de objetos colisionantes y se
empieza otra vez*/
                                         block=COLLISION(type block);
                              end
                    end

                    //Si la bola choca contra la raqueta...
                    if(paddle!=0)
                              vy=-vy;//...invertimos la velocidad vertical...
/*...y alteramos la velocidad horizontal dependiendo de en qué punto de la raqueta se
haya colisionado..*/
                              vx=vx+(x-paddle.x)/2;
                    end
//Si la bola choca contra los límites de la pantalla (excepto el inferior)
                    if(x<=2 or x>=318)vx=-vx;end
```

```
                        if(y<=2)vy=-vy;end

//Limitamos la velocidad horizontal un poco (para que no se embale mucho)...
                        if(vx>20)vx=20;end
                        x=x+vx;//y la bola se mueve
                        y=y+vy;
                        if(y>260)return;end //Si perdemos la bola, matamos el proceso
                        frame;
            end
end
```

## *La variable local predefinida RESOLUTION

Otra variable local predefinida importante es ***RESOLUTION***. Esta variable establece el factor de ampliación (ó mejor dicho, precisión) para las coordenadas ***X*** e ***Y*** del proceso en cuestión. Es decir, permite aumentar la resolución de las coordenadas ***X*** e ***Y*** del proceso, ya que indica cuántas unidades lógicas existen por cada píxel de pantalla. Es decir, la variable ***RESOLUTION*** hace que las coordenadas de un proceso concreto se interpreten multiplicándose siempre por el valor que tenga ella misma, así que cuando por ejemplo se muestre el gráfico de un proceso, aparecerá como si la posición ***X*** e ***Y*** estuvieran dividido por el valor de ***RESOLUTION***. Dicho de otra manera:avanzar 400 unidades (con **advance()**, por ejemplo) en un proceso con resolución 100 nos desplaza únicamente 4 píxels en la pantalla.

Es conveniente usar esta variable para ralentizar el movimiento de objetos que deban moverse lentamente o para aumentar la suavidad en movimientos complejos que depende de la precisión en calculos matemáticos como tiros parabólicos, gravedad, etc. Por defecto su valor es 0, indicando que la equivalencia es de 1:1. También puede tener valores negativos; en ese caso, en vez de ser un factor de ampliación de coordenadas, es de división, produciéndose el efecto contrario.

El siguiente código se puede apreciar cómo, cambiando el valor de ***RESOLUTION*** se produce (o no), un suavizado en el dibujado de la cola que persigue a la bola. A más resolución, mayor será el suavizado. No importa si aparecen órdenes y elementos de código que no hemos visto todavía: lo único en lo que te tienes que fijar es en observar cómo cambia el resultado del movimiento de la cola que persigue a la bolita en la pantalla según se cambie el valor de ***RESOLUTION***

```
import "mod_grproc";
import "mod_time";
import "mod_key";
import "mod_video";
import "mod_map";
import "mod_draw";
import "mod_proc";
import "mod_wm";

Process Main()
Begin
    set_mode(320,200,32);
    set_fps(60,0);

//Dice la resolución de este proceso. Cambia su valor(10,50,100...)para ver el efecto
    resolution = 1;

    graph = map_new(200,200,32);drawing_map(0,graph);
    drawing_color(rgb(0,255,255)); draw_fcircle(100,100,99);
    size = 10;

    x = 160 * resolution;
    y = 180 * resolution;

    Repeat
        //Creo una minicola de este proceso durante un segundo, a cada fotograma
        trail(x,y,graph,get_timer()+1000);
        advance(3*resolution); //Avanza (3*resolution) unidades. Es decir, 3 píxeles
```

```
        angle=angle + 2000; //Giro en círculos, dejando la minicola detrás
        frame;
    Until(key(_ESC)||exit_status)

OnExit
    let_me_alone();
    unload_map(0,graph);
End

Process trail(x,y,graph,int endtime)
Begin
    //Recoge la resolución de su proceso padre
    resolution = father.resolution;
    size=father.size/5;

    //Existe hasta que llega un determinado tiempo final
    Repeat
        frame;
    Until(get_timer()>=endtime)

End
```

# CAPITULO 7

## Tutorial para un matamarcianos

El matamarcianos que se desarrollará en este tutorial es tremendamente parecido al juego del laberinto presentado en un capítulo anterior, con unas pocas alteraciones. Recomiendo leer pues aquel tutorial antes para poder seguir el tutorial presente, ya que éste se puede considerar una síntesis resumida –con algunos añadidos mínimos- del juego del laberinto, y por tanto no se volverán a explicar conceptos que ya hayan aparecido anteriormente.

### *Punto de partida

Empezaremos dibujando un fondo estrellado –de color azul oscuro con puntitos blancos- que representará el universo, y una nave espacial, que será el protagonista que controlaremos, de unos 70x70 píxeles, y mirando hacia arriba. Una vez que tengamos las imágenes, las añadiremos a un FPG llamado "prueba.fpg": el fondo con código 001 y la nave con código 002, y comenzaremos nuestro programa escribiendo esto (se explica por sí solo):

```
import "mod_video";
import "mod_map";
import "mod_screen";
import "mod_key";

Global
          Int Graficos;
End

Process Main()
Begin
          set_mode(640,480);
          set_fps(60,1);
          Graficos=load_fpg("prueba.fpg");
          Put_screen(graficos,1);
          Nave();
          Loop
                    If (key(_esc)) break; end
                    Frame;
          End
          Unload_fpg(graficos);
End

Process nave ()
Begin
//No es necesario utilizar la variable FILE porque sólo hemos cargado un FPG
          Graph=2;
          X=320;
//La nave está a 10 píxeles del extremo inferior de la ventana
          Y=435;
          Loop
                    If (key(_left)) x=x-4; end
                    If (key(_right)) x=x+4; end
                    If (x>600) x=600; end //Para que la nave no se salga por la derecha
                    If (x<40) x=40; end //Para que la nave no se salga por la izquierda
                    Frame;
          End
end
```

Lo primero que añadiremos al código es la orden **let_me_alone();**, que la escribiremos justo antes del end final del programa principal (después del *unload_fpg(graficos);*), para que cuando salgamos del bucle de éste, matemos todos los procesos que pudieran estar activos en ese momento (de momento sólo tenemos el proceso "Nave()", pero en

seguida pondremos otros, como "Enemigo()" o "Disparo()"), y podamos acabar el programa limpiamente y sin problemas,una vez se haya acabado de ejecutar el código del programa principal.

## *Añadiendo los disparos de nuestra nave. La variable local predefinida Z

Seguidamente, haremos el dibujo del disparo, de unos 10x30 píxeles. El código para guardarlo en el FPG será el 3. Los disparos son un nuevo elemento con autonomía propia dentro del juego, por lo que tendremos que crear un nuevo proceso para ellos: el proceso "disparo()".

```
Process disparo ()
Begin
          Graph=3;
          Loop
                    y=y-15;
                    frame;
          end
end
```

Puedes ver que a cada frame, la posición vertical decrece en 15 píxeles: así pues, en cada frame el disparo va subiendo por la pantalla 15 píxeles.

La posición vertical del disparo al inicio la conocemos, siempre será y=435, ya que nuestra nave nunca se va a mover hacia arriba ni hacia abajo, pero ¿y la coordenada x? Depende de la posición de la nave, por lo que usaremos el mismo truco de siempre: los parámetros. Por ahora vamos a dejar esto en suspense, vamos a decirle primero al programa cuándo tiene que crear un disparo.

El disparo se creará siempre que se pulse la tecla de disparo, pero ¿cuál crees que será el mejor sitio para crearlo dentro del código? Como el disparo depende directamente de la nave protagonista, haremos que sea este proceso el que se encargue de crear el disparo. Usaremos para ello la tecla “x” como tecla de disparo. Por lo tanto, añadiremos la siguiente línea dentro del bucle LOOP del proceso "nave()", justo antes de la orden frame:

```
if (key(_x)) disparo(); end
```

De esta forma se creará un disparo, siempre que se pulse la tecla adecuada.

Volvamos al proceso “disparo()”. Decíamos que la posición de origen del disparo dependía de la nave... Si te acuerdas del capítulo anterior en este manual, para hacer que las explosiones aparecieran en la misma posición que nuestro personaje, lo que hacíamos era que el proceso explosión recibiera por parámetro los valores de ***X*** e ***Y*** del personaje. Recuerda que esto se hacía en dos pasos: en la llamada al proceso "explosion()" -que recordemos que se hacía dentro del código de nuestro personaje-, poníamos como valores de sus parámetros no ningún número en concreto, sino los valores que tenían en ese momento la ***X*** e ***Y*** del personaje: es decir, escribíamos: "*explosion(x,y);"*. Y luego, en la cabecera del proceso "explosion()", hacíamos que esos valores pasados fueran asignadosos precisamente a la ***X*** e ***Y*** de la explosión: es decir, la cabecera del proceso explosión la escribíamos así: "*process explosion(x,y)*". Podríamos hacer ahora exactamente lo mismo: nuestro personaje ahora será el proceso "nave()" y lo que en el ejemplo anterior era una explosión, ahora sería el proceso "disparo()". Pero lo vamos a hacer de otra forma para que veas otras posibilidades (aunque es bastante similar,de hecho)

Tal como tenemos escrito nuestro juego hasta aquí, ahora existe una relación entre ambos procesos: “nave()” ha creado a “disparo()”, por lo que “nave()” es el Father de “disparo()” y por tanto “disparo()” es Son de “nave()”, y dos disparos son hermanos, porque tienen el mismo padre. Usaremos este hecho para utilizar las variables locales predefinidas ***FATHER*** y ***SON*** convenientemente. Modifica así el proceso disparo así:

```
Process disparo ()
Begin
          Graph=3;
          Y=435;
          X=father.x;
          Loop
                    y=y-15;
```

```
                    frame;
          end
end
```

Ahora le estamos diciendo que se ponga en la misma coordenada ***X*** que su padre, (es decir, la nave) utilizando la sintaxis típica para acceder a los valores de las variables locales de procesos: *identificadorProceso.variableLocal*. No obstante, hay que tener en cuenta que este sistema sólo lo podremos utilizar tal como está ahora si sabemos que el proceso disparo tiene como único Father posible la nave. Si quisiéramos por ejemplo reutilizar el proceso "disparo()" para que nuestros posibles enemigos pudieran disparar también, entonces la variable ***FATHER*** valdría para cada disparo en particular uno de dos valores posibles (proceso "nave()" o proceso "enemigo()" y entonces la ***X*** del disparo cambiaría según de qué padre proviniese.Ojo.

Si ejecutas ahora el juego tal como está, verás que tu nave dispara por fin, pero quizás no te guste el detalle de que los disparos aparezcan por encima de la nave y desde el centro. Modificamos pues el proceso "disparo()" de la siguiente manera:

```
Process disparo ()
Begin
          Graph=3;
          Y=410;
          X=father.x;
          Z=1;
          Loop
                    y=y-15;
                    frame;
          end
end
```

Lo que hemos hecho ha sido modificar la coordenada ***Y*** del origen del disparo, poniéndola un poco más arriba, ya que tenemos la suerte de que la nave sólo se moverá horizontalmente, y por tanto, podemos fijar que los disparos aparecerán a partir de Y=410. Pero si la nave se pudiera mover también verticalmente, ¿que ***Y*** tendríamos que haber puesto? Bueno, al igual que hemos recogido la posición horizontal de la nave para asignársela a la del disparo, igualmente habríamos hecho con la posición vertical, procurando restarle unos determinados píxeles para que el disparo continuara saliendo más arriba que la nave. Es decir:

```
y=father.y-25;
```

Existiría otra posibilidad diferente y es que la nave pudiera rotar (es decir, pudiera cambiar su ***ANGLE***). En este caso, los disparos tendrían que salir en la misma dirección en la que la nave estuviera encarada en cada momento. Lo primero que tendríamos que escribir en esta nueva situación en el proceso "disparo()" (antes del LOOP) es:

```
x=father.x;
y=father.y;
angle=father.angle;
```

(o bien, también serviría pasar estos tres valores como parámetro, tal como se discutió en el tutorial del juego del laberinto en el apartado de las explosiones) y, también muy importante, además tendríamos que cambiar la línea de dentro de su LOOP/END que decrementaba la y (*y=y-15;*) y poner en su lugar *advance(15);*, de tal manera que los disparos se movieran en la misma dirección a la que está mirando la nave en ese momento (lógicamente). No obstante, haciéndolo de esta manera volveríamos a tener el mismo problema de antes: el disparo aparece por encima de la nave, y ahora no podemos hacer el truco de restar una cantidad a la coordenada vertical porque la nave, si puede rotar, no necesariamente estará mirando "hacia arriba": quedaría fatal que la nave estuviera mirando al noroeste por ejemplo y los disparos salieran 25 píxeles desplazados por encima de la nave , creándose de la nada. En este caso, el truco estaría en escribir justo antes (fuera) del bucle LOOP/END del proceso disparo, otra línea *advance(15);* . Es decir, que antes de que empieze el bucle se ejecute "de serie" un *advance(15);* Lo que hace esta línea es forzar a que el disparo se mueva antes de empezar el LOOP de su proceso, con lo que cuando empiece el bucle el disparo ya no estará en el centro, sino "delante" del personaje. De esta manera, el primer *advance(15);* que se ejecute de dentro del LOOP/END en la primera iteración se acumulará al *advance(15);* de fuera, y así, cuando se llegue al primer frame, en realidad el disparo avanzará de golpe 30 píxeles, con lo que habremos solucionado el problema.

La otra modificación que se ha hecho ha sido añadir la línea *Z=1;*.Fíjate que si no se pone, los disparos se

pintarán por encima de la nave, cosa que queda feo. ***Z*** es otra variable local predefinida que indica el nivel de profundidad de un proceso dado en relación al resto. Esto sirve para lo que has visto: pintar unos procesos por encima de otros, bien ordenaditos en la tercera dimensión: a mayor valor de ***Z***, más lejano se pintará el proceso; a menor valor de Z, más cercano estará, siendo el rango posible de valores entre -512 y 512.¿Y por qué los disparos, si no se especifica ningún valor de ***Z***, aparecen dibujados sobre la nave? Por una razón que has de recordar siempre a partir de ahora: en principio, el último proceso en ser creado será el que se vea por encima de los demás. Es decir, si no se especifica lo contrario, los procesos se ordenan en profundidad en función de su momento de creación (procesos posteriores, **Z** más cercana). En nuestro código está claro que cualquier proceso "disparo()" se ha creado después del proceso "nave()", porque entre otras cosas, si no hay proceso nave creado previamente, los disparos no pueden existir., por lo que, siguiendo esta regla, cualquier proceso disparo aparecerá pintado sobre la nave. Para evitarlo, tendremos que cambiar la profundidad a la que aparece el gráfico del disparo,especificando para este proceso un determinado valor de esta variable local. ¿Y cómo sabemos que tenemos que poner un 1? Porque, otra cosa que tienes que saber, es que todos los procesos creados en el programa, por defecto siempre tienen el valor de ***Z*** igual a 0. Así pues, el proceso "nave()" sabemos que tiene una Z=0, pero ojo, los procesos "disparo()" también, si no se le dice lo contrario. El tema está lo que acabo de comentar en las frases anteriores: aun cuando existan varios procesos con ***Z*** iguales, el proceso de pintado es el siguiente: tendrá mayor profundidad aquel proceso que haya sido creado antes. Es decir, a medida que se van creando procesos, éstos se van "amontonando" encima de los ya existentes (repito, aún teniendo una Z igual). Por lo tanto, si queremos que los disparos se pinten bajo el proceso nave, tendremos que obligarles explícitamente a que que su Z sea MAYOR que 0, que es el valor que tiene la ***Z*** de "nave()" por defecto. Con 1 ya vale. Recuerda: cuanto más negativa es la ***Z***, más hacia el frente se vienen los gráficos, y a ***Z*** más positivas, más hacia el fondo se pintan.

Es conveniente saber qué valores predeterminados puede valer la ***Z*** según el tipo de gráficos que aparezcan en pantalla. Lo más al fondo que hay son los scrolls y los gráficos pintados con las funciones de la familia "put" (como "**put_screen()**"), que por defecto tienen una Z=512. Después están los planos abatidos Modo7, con una Z=256 por defecto. Le siguen los gráficos de los procesos (ya lo hemos dicho, con Z=0), luego los textos (con Z=-256) y por último, lo que se pinta por encima de todo, el gráfico del cursor del ratón (Z=-512). Según avancemos veremos que estos valores se pueden modificar, como ya hemos hecho con un gráfico de un proceso.

Puedes probar un momentito este ejemplo (saliéndonos momentáneamente de nuestro matamarcianos) para ver mejor lo que te cuento sobre la ***Z***. El programa principal creará tres procesos, cada uno de los cuales será visualizado mediante un cuadrado de diferente color, que se podrán mover mediante diferentes teclas del cursor. También mostrará el valor por defecto de ***Z*** para los tres procesos, que será 0. Y ya está. Lo que quiero que veas es que, aún teniendo la misma ***Z***, el cuadrado correspondiente a "proceso1()" -el rojo- estará a menor profundidad que los otros dos, es decir, se visualizará por encima de los otros dos (lo puedes comprobar si lo mueves y lo pasas por encima de los otros gráficos); el cuadrado de "proceso2()" -el verde- estará sobre el de "proceso3()" pero bajo el de "proceso1()" y el de "proceso3()" -el azul- se pintará bajo los otros dos. Esto es debido al orden de creación de los procesos: primero se ha creado "proceso1()", luego "proceso2()" y luego "proceso3()". Si quieres, puedes cambiar este orden y comprobarás que las profundades cambian también – a pesar de continuar teniendo los mismos valores de ***Z*** todos los procesos-. Ahora bien, si definieras explícitamente un valor de ***Z*** para estos procesos, sería todo más claro, porque controlarías exactamente a qué profundidad respecto el resto deseas pintar cada proceso.

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";

process main()
private
          int idproc1,idproc2,idproc3;
end
begin
          idproc1=proceso1();
          idproc2=proceso2();
          idproc3=proceso3();
          write_var(0,100,20,4,idproc1.z);
          write_var(0,100,40,4,idproc2.z);
          write_var(0,100,60,4,idproc3.z);
          loop
                    if(key(_esc)) exit(); end
                    frame;
          end
end
```

```
Process proceso1()
begin
          graph=new_map(50,50,32);map_clear(0,graph,rgb(255,0,0));
          x = 160;
          y = 120;
          loop
               if(key(_up))y=y-5;end
               if(key(_down))y=y+5; end
               if(key(_left))x=x-5; end
               if(key(_right))x=x+5; end
       frame;
   end
end

Process proceso2()
begin
          graph=new_map(50,50,32);map_clear(0,graph,rgb(0,255,0));
          x = 60;
          y = 20;
          loop
               if(key(_w))y=y-5;end
               if(key(_s))y=y+5; end
               if(key(_a))x=x-5; end
               if(key(_d))x=x+5; end
       frame;
   end
end

Process proceso3()
begin
          graph=new_map(50,50,32);map_clear(0,graph,rgb(0,0,255));
          x = 260;
          y = 220;
          loop
               if(key(_h))y=y-5;end
               if(key(_n))y=y+5; end
               if(key(_b))x=x-5; end
               if(key(_m))x=x+5; end
       frame;
   end
end
```

Por cierto, una cosa importante que deberás recordar: el valor de ***Z*** no se tiene en cuenta para la detección de colisiones. Es decir, las colisiones se pueden producir entre diferentes procesos aunque éstos estén en ***Z*** diferentes. Cuando añadamos enemigos a nuestro juego (dentro de nada) verás que los enemigos estarán a otra ***Z*** diferente de los disparos, pero aun así, los disparos colisionarán perfectamente con éstos, ya que, como hemos dicho, la tercera dimensión es irrelevante en la detección de colisiones.

Volvamos al matamarcianos. Te habrás puesto a disparar como un loco y habrás comprobado que a medida que pasa el tiempo y disparas más parece que el juego se ralentiza, puede que tardes diez segundos o cinco minutos, pero llega un momento que la cosa se vuelve injugable ¿Qué está pasando? La respuesta es bien sencilla, el disparo, aunque salga de pantalla aun sigue activo, y sigue avanzando hasta que llegue al infinito, y claro, si el primer disparo sigue ejecutándose tras el 100º disparo significa que el ordenador está moviendo 100 procesos, y llega un momento en que tiene que ejecutar demasiados procesos y no le da tiempo. Es nuestro trabajo evitarlo, tenemos que hacer que el disparo se autodestruya al salir del área visible. Hay dos formas de hacerlo, una es usando la función **out_of_region()**, de la que encontrarás más información posteriormente, y la otra es la que vamos a usar, que es bastante más sencilla: simplemente se trata de comprobar si el disparo está por encima de la posición vertical –15, situación en la que se sale del bucle. Esto lo haremos añadiendo un IF así:

```
Process disparo ()
Begin
          Graph=3;
          Y=410;
          X=father.x;
          Z=1;
```

```
        Loop
                y=y-15;
                if (y < -15) break; end
                frame;
        end
end
```

Haz la prueba y verás que ha mejorado notablemente, puedes tirarte horas disparando y la munición no se acabará, y el rendimiento no se verá afectado.

No obstante, quizás podamos mejorar ligeramente el rendimiento todavía un poco más. Esto lo haremos usando otro tipo de bucle más apropiado dado que sabemos cuando va a dejar de ejecutar éste. Tan sencillo como sustituir el bucle LOOP por REPEAT/UNTIL:

```
Process disparo ()
Begin
        Graph=3;
        Y=410;
        X=father.x;
        Z=1;
        Repeat
                y=y-15;
                frame;
        until (y < -15)
end
```

Este bucle es más apropiado ya que, al menos, se ejecutará una vez pero se acabará cuando supere el borde superior de la pantalla. Quizás te preguntes por qué debe superar –15 y no 0, y eso se debe a que las coordenadas del gráfico están en el centro, y de borrarlo en cero aun se vería la mitad del disparo.

## *Añadiendo los enemigos

Abre tu editor de imagen preferido y diseña una nave enemiga del mismo tamaño que la protagonista, pero mirando hacia abajo, con código de gráfico número 4.

¿En qué lugar del código crearías los enemigos? Por el momento tenemos tres procesos. En el proceso "disparo()" no es, porque si no disparas no saldría ninguna nave enemiga. El proceso "nave()" no tiene este problema, pero no es el más adecuado, imagínate, el bueno creando el mal. Así que nos quedamos en el proceso principal.

Ahora hay que saber donde meterlo, lo lógico es que esté en el bucle principal, de lo contrario sólo se crearía un enemigo, así que vamos a añadir la línea *enemigo();* dentro del LOOP del programa principal, justo antes de la orden FRAME.Y a más a más, claro, escribiremos después de todo lo que hemos escrito hasta ahora el código de este nuevo proceso:

```
Process enemigo ()
Begin
        Graph=4;
        Y=-40;
        Repeat
                Y=y+5;
                Frame;
        Until (y > 520)
end
```

Ves que lo que van a hacer nuestros enemigos será aparecer desde fuera de la pantalla por arriba, y avanzar hacia la parte inferior (5 píxeles) y desaparecer por abajo. Pero también tendremos que, primero, hacer que se muevan de lado para que sea más difícil acertarles, y, segundo, hacer que cada enemigo aparezca por arriba en una X diferente de los demás. Vayamos por lo segundo primero.

Ya sabemos que cuando necesitamos crear procesos con unas condiciones que varían según el momento,

una solución a la que podemos recurrir es el paso de parámetros, o sea, indicarle al ordenador que cuando cree un proceso le indique de qué forma hacerlo. En nuestro caso, vamos a crear el proceso enemigo indicándole qué posición horizontal –fija y común, de momento- debe tener al ser creado, y lo haremos invocándolo en el proceso principal en vez de con la línea *enemigo();* , con la línea siguiente:

```
enemigo(320); //Indico un valor concreto del parámetro
```

Y el código del proceso "enemigo()" quedaría así:

```
/*El valor pasado en el programa principal se  asigna a la variable local X del proceso
enemigo  (por lo que todos los enemigos saldrán en el  centro de la pantalla.*/
Process enemigo (x)
Begin
          Graph=4;
          Y=-40;
          Repeat
                    Y=y+5;
                    Frame;
          Until (y > 520)
end
```

Si compilas y ejecutas ahora, verás que aparecen un montón de naves, todas en fila, bajando por el centro de la pantalla. La verdad es que seguramente no era lo que esperabas, tu querrás que aparezcan en distintas posiciones, pero ¿cómo lo hacemos? Si ponemos un número siempre aparecerán por esa posición. La solución –una de ellas- está en generar números aleatorios, al azar. Para eso usaremos la conocida función **rand()**, que viene de la palabra inglesa "random" (aleatorio). Modifiquemos pues otra vez la llamada al proceso "enemigo()" dentro del programa principal para que en vez de pasarle como parámetro el número fijo 320, sea un número aleatorio, así:

```
enemigo(rand(0,640));
```

Si haces la prueba obtendrás una auténtica lluvia de enemigos, un auténtico aluvión. Quizás hay demasiados enemigos ¿no? La pantalla está saturada y casi no se ve el espacio. Vamos a arreglarlo modificando el código del proceso principal de la siguiente manera:

```
Process Main()
Begin
          set_mode(640,480);
          set_fps(60,1);
          Graficos=load_fpg("prueba.fpg");
          Put_screen(graficos,1);
           Nave();
          Loop
                    If(rand(0,100)<20)
                              enemigo(rand(0,640));
                    End
                    If (key(_esc)) break; end
                    Frame;
          End
          Unload_fpg(graficos);
         Let_me_alone();
End
```

Así generamos un número entre 0 y 100, y sólo cada vez que este número sea menor que veinte se creará una nave enemiga, es decir, que en cada frame hay un 20% de posibilidades (20 de 100) de que aparezca un nuevo enemigo, de esta manera se creará una nave cada 5 frames como media (como son datos aleatorios, a veces serán más y otras menos). Puedes ejecutar el programa y ajustar el valor a tu gusto.

Ya que le hemos pasado como parámetro desde donde empieza, vamos a aprovechar para pasarle otros parámetros, por ejemplo, cuánto debe avanzar lateralmente y verticalmente (por lo que hay una línea del proceso enemigo que hay que modificar), empecemos por modificar en el proceso principal la llamada al proceso enemigo, pasándole en vez de uno, tres valores como parámetro, así:

```
enemigo(rand(0,640), rand(-5,5),rand(4,7));
```

Evidentemente, si llamas a un proceso con tres parámetros, éste debe recibir tres parámetros, y por eso en la cabecera deben figurar tres variables, y no una como tenemos ahora, así que hagamos hay que hacer, antes de hacer nada más, la siguiente modificación en el proceso enemigo:

```
Process enemigo (x,int inc_x,int inc_y)
Begin
          Graph=4;
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    Frame;
          Until (y > 520)
end
```

Lo que hemos hecho ha sido asignar dos números aleatorios a dos variables privadas creadas por nosotros (*inc_x* e *inc_y*) de manera que así se desplacen de forma distinta. Horizontalmente, -la coordenada x-, se podrán desplazar tanto a izquierda como a derecha, y por eso el número aleatorio puede ser tanto positivo como negativo. Verticalmente, -la coordenada y-, sólo se desplazan hacia abajo, por eso son números positivos mayores de cero. Explicado más detenidamente: antes de pintar la nave la movemos hacia un lado modificando su variable ***X*** el número de píxeles que valga *inc_x* (ya sea a la izquierda, por ser un valor negativo, a la derecha, por ser un número positivo, o no se moverá de lado al valer cero), luego la movemos hacia abajo (lo que valga ***inc_y***, que cuanto mayor sea, más rápido se moverá) y por último la pintamos en la nueva posición, y a repetir otra vez hasta que salga de la pantalla. Si lo ejecutas ahora, la lluvia ya no se desplaza sólo hacia abajo, ya se mueven a distinta velocidad y en diversas direcciones.

Ya que estamos, podemos hacer que los enemigos sean distintos entre sí, bien metiendo nuevas gráficos al FPG y usando **rand()** en la variable ***GRAPH*** del proceso "enemigo()", o también haciendo que sean de distinto tamaño. Una ventaja de estos programas con respecto a otros lenguajes es que esto último podemos hacerlo de manera casi automática, gracias a la variable local ***SIZE*** -"tamaño" en inglés-. Esta variable, como ya sabemos,por defecto (es decir,si no la cambias), vale 100, e indica el porcentaje del tamaño de la imagen. Si le asignas el valor 50, el gráfico se verá al 50% de su tamaño, es decir, la mitad de grande Así que escribiremos:

```
Process enemigo (x,int inc_x,int inc_y)
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    Frame;
          Until (y > 520)
end
```

## *Matando los enemigos (eliminando procesos)

La muerte de los enemigos hay dos formas de tratarla, desde los disparos o desde los propios enemigos, ya que son los procesos que están interactuando. Aunque se podría hacer al revés, lo más lógico es que sea el propio enemigo el que decida si debe morir o no, porque es posible que un disparo no sea suficiente para eliminarlo. En nuestro caso, para no complicarnos mucho, el enemigo morirá al más leve roce de tu munición, así que ya tenemos una nueva condición para acabar con el bucle.

El problema es que cómo sabe un enemigo concreto si está chocando con un disparo si en pantalla hay como unos 25. Podríamos calcular la distancia a cada uno, pero es que no sabemos los números de identificación de todos ellos, y aunque los guardáramos, habría que ver de qué manera lo haríamos porque no sabríamos cuántos disparos habría ni los podríamos almacenar en un mismo sitio...Parece que la cosa está complicada, pero por suerte nuestros lenguajes están ahí para facilitarnos las cosas: la palabra clave es, como podías esperar, **collision()**. Recuerda que esta función detecta si el proceso que usa esta función está chocando con el que le pasamos por parámetro, es decir, si algún pixel NO TRANSPARENTE del proceso que usa **collision()** se superpone con otro pixel NO TRANSPARENTE del que le decimos.

Pero ahora tenemos el problema de tener que preguntar en cada enemigo por todos los procesos disparo uno a uno... Pues no, la cosa se soluciona igual de fácilmente, como ya sabrás, usando TYPE más el nombre del proceso. Así estamos diciendo que nos referimos a cualquier proceso de ese "tipo", en nuestro caso "disparo()". Cojamos el proceso "enemigo()" y pongámonos a escribir:

```
Process enemigo (x,int inc_x,int inc_y)
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    Frame;
          Until (y > 520 OR collision(type disparo))
end
```

De esta forma, cuando el enemigo choque con un disparo, se saldrá del bucle y desaparecerá de pantalla porque **collision()** devolverá un valor "true" (verdadero). Date cuenta de un detalle importante, y es que después de "disparo()" no hemos puesto los paréntesis, porque si los ponemos le estamos diciendo al ordenador que cree un proceso disparo (y nos pedirá los parámetros). Al poner OR en medio de ambas posibilidades dentro del "Until", se debe cumplir al menos una de ellas para salir, pues recuerda el funcionamiento de la operación OR; si hubiéramos querido que se saliera cuando ambas fueran verdaderas hubiéramos usado AND.

Otra cosa que quizás no te guste es que los enemigos desaparecen al contacto con el disparo, sin fuego ni sangre ni un mal ¡puf! No te preocupes por ahora, eso lo solucionaremos más adelante, pues antes de ponernos con los detalles "decorativos" debemos preocuparnos del aspecto "funcional", esto significa, que si no funciona, de nada nos sirve hacer la explosión si después hay que cambiarlo todo.Pero ¿qué falla? Parece que todo va bien. Sí, la cosa va bien, pero hay algo que no cuadra.

Si un disparo choca con la nave enemiga, el disparo sigue como si nada. Para evitar esto, se podría poner una sentencia en el proceso "disparo()" que lo finalizara al chocar con el enemigo, pero no suele funcionar: cuando el proceso enemigo choca con el disparo ejecuta su propia sentencia **collision()** y por lo tanto desaparece, por lo que cuando el proceso "disparo()" llega a su propia sentencia **collision()** ya no está chocando con el enemigo y no desaparece, y viceversa. No hay manera que desaparezcan los dos a la vez. La mejor manera que he encontrado para solucionar esto es usar códigos identificadores

Recuerda que todos los números de identificación de los procesos (las ID, o los números que devuelven al ser llamados) son diferentes de 0. Así que si estamos asignando ese valor a una variable, esta valdrá siempre "true". *Y aprovecho para decir que esto es aplicable también a los ficheros (FPG) y gráficos (PNG), de manera que sabremos en cualquier momento si se ha cargado o no (si no se carga vale 0, y si se carga valdrá otro número, es decir, "true").*Y tras este rollo, lo que veníamos a decir, que ya lo dije en el tutorial del juego del laberinto: **collision()** no devuelve "verdadero" simplemente, sino la ID del proceso con el que colisiona.

Esto no es importante si el parámetro que le hemos pasado es una ID que sabíamos de antemano (el proceso "nave()" u otro en concreto), pero en nuestro caso, que puede ser un proceso cualquiera, del tipo que le hemos pedido, del que hay varios "iguales", es utilísimo, ya que nos dice, sin margen de error, quién es el responsable que está chocando con él. De esta forma, podemos saber cuál es el disparo que debemos eliminar para que no siga adelante. Lo haremos así, asignando la ID que devuelve a una variable, comprobaremos si alguno ha colisionado, y actuaremos en consecuencia "matando" al proceso:

```
Process enemigo (x,int inc_x,int inc_y)
Private
          Int ID_disparo_acertado;
end
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
```

```
                    ID_Disparo_acertado=collision(type disparo);
                    If (ID_disparo_acertado !=0)
                              Signal(ID_disparo_acertado,s_kill);
                    End
                    Frame;
          Until (y > 520 OR ID_Disparo_acertado !=0)
end
```

Parece que hemos realizado bastantes cambios, vamos a analizarlos. Hemos creado una nueva variable, (privada, y por tanto no se podrá leer ni modificar desde otro proceso), para almacenar el ID del disparo que choca con nuestra nave, este valor valdrá cero mientras no haya una colisión. Una vez almacenado el dato comprobamos si ha habido colisión, y en caso afirmativo hacemos que desaparezca el disparo, para ello usamos la función conocida función **signal()**. Recuerda que esta instrucción permite mandarle "señales" a otros procesos para que reaccionen de una forma u otra, para ello necesitamos dos parámetros: el primero es la ID del proceso al que le vamos a mandar la señal, cosa que no es problema en nuestro caso, y el segundo es la señal que le vamos a mandar (recuerda: S_KILL, S_SLEEP,S_FREEZE, S_WAKEUP, S_KILL_TREE, S_SLEEP_TREE,S_FREEZE_TREE ó S_WAKEUP_TREE).

Por último, comprobamos si se cumple cualquiera de las condiciones para que el enemigo desaparezca, bien por salir de pantalla o por haber recibido un impacto directo gracias a tu grandiosa puntería. Si se cumple, el enemigo se autodestruye saliendo del bucle REPEAT/UNTIL. De esta manera nos aseguramos de que desaparecerán tanto el proceso enemigo como el disparo y evitamos el problema que teníamos antes de que los disparos continuaban después de colisionar con el enemigo.

También podríamos haber escrito, de forma alternativa, el código del enemigo así, ahorrándonos de escribir alguna línea:

```
Process enemigo (x,int inc_x,int inc_y)
Private
          Int ID_disparo_acertado;
end
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    If (ID_Disparo_acertado=collision(type disparo) !=0)
                              Signal(ID_disparo_acertado,s_kill);
                    End
                    Frame;
          Until (y > 520 OR ID_Disparo_acertado !=0)
end
```

Analicemos la condición con cuidado. Se hace el mismo IF, pero con un cambio: el paréntesis tiene una asignación además de la comparación. El orden de procesamiento siempre es: 1º la asignación y 2º la comparación Por lo tanto, primero se asigna a "ID_Disparo_acertado" el valor de retorno de la función **collision()** (que ya sabemos que será 0 si no hay colisión o el ID del disparo concreto si la hay), y seguidamente, en la misma línea, se compara este valor de *ID_Disparo_acertado* para ver si es diferente de 0. Es otra manera de escribir lo mismo, más compacto.

Vamos a ponernos ahora un momento en la piel de un programador. Algo muy importante a la hora de hacer un programa es que este sea "eficiente": esto es, que en el menor tiempo posible se haga la mayor cantidad de trabajo posible, y esto incluye evitar repetir las operaciones o usar sólo la memoria imprescindible. Esto se consigue usando el menor número de variables del tipo más adecuado, usando procesos sin gráficos para ejecutar operaciones que se realizan a menudo (normalmente a estos procesos que "no se ven" se les llama funciones o sub-rutinas), evitando repetir operaciones que se pueden resolver con bucles, y siendo ya muy tiquismiquis, evitando acceder a memoria, sobre todo secundaria (disco duro, disquetes, CD ROM) ya que es lo que consume más tiempo. Del otro lado tenemos el orden del código: si usas una variable para siete cosas distintas al final no sabrás lo que está haciendo en cada momento, por ejemplo, usar *var1* como contador, como variable auxiliar, para ver la energía…: reduce pero con moderación. Por ahora tampoco debe preocuparte demasiado, pero sí debes tener en cuenta que la carga de archivos (imágenes, música...) tarda mucho tiempo y consume memoria, por lo que deberías cargar siempre sólo las imágenes que vayas a usar en cada momento (de aquí la importancia de saber agrupar las imágenes en los ficheros).

Bueno, supongo que mientras has estado leyendo el párrafo anterior habrás pensado en nuestro proceso y te habrás dado cuenta de que hay un problema de eficiencia: hacemos la comprobación de si el enemigo ha chocado dos veces (una en el IF y otra en el UNTIL) y esto hay que remediarlo ¿cómo? Bueno, la segunda comprobación sirve para salir del bucle... ¿no te suena de algo? claro, nuestra instrucción BREAK, si la metemos dentro de las instrucciones del IF ya no hace falta comprobarlo de nuevo. Hemos ganado en eficiencia evitando una comprobación (e incluso el posterior FRAME y la comprobación del UNTIL para salir).

```
Process enemigo (x,int inc_x,int inc_y)
Private
          Int ID_disparo_acertado;
end
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    ID_Disparo_acertado=collision(type disparo);
                    If (ID_disparo_acertado !=0)
                              Signal(ID_disparo_acertado,s_kill);
                             break;
                    End
                    Frame;
          Until (y > 520)
end
```

Considerarás conmigo que no es justo que nuestro protagonista pueda disparar a diestro y siniestro: a la larga hace nuestro juego aburrido. Podemos reducir el número de disparos que podemos hacer usando un contador que, mientras no sea cero, no pueda disparar. Pero para controlar el retardo ¿qué valor debemos darle? ¿5? ¿10? En realidad no lo sabemos y sólo podemos hacerlo a ojo, y para hacerlo vamos a crear una constante, de tal manera que sólo tenemos que cambiarle el valor en esa zona para controlarlo, y así no tenemos que leer todo el código buscando todas las referencias a él (y lo más seguro es que nos equivoquemos cambiando algo). Cojamos el programa principal y añadamos la declaración de una constante llamada “Retardo”, justo después de las líneas "import":

```
Const
          Int Retardo=10;
End
```

Y a continuación vamos a hacer los cambios en el proceso "nave()", que es el que se encarga de crear los disparos:

```
Process nave ()
Private
          Int cont=0;
End
Begin
          Graph=2;
          X=320;
          Y=435;
          Loop
                    Cont=cont-1;
                    If(cont<0) cont=0; end
                    If (key(_left)) x=x-4; end
                    If (key(_right)) x=x+4; end
                    If (x>600) x=600; end
                    If (x<40) x=40; end
                    If (key(_x) AND cont==0)
                              cont=retardo;
                              disparo();
                    end
                    Frame;
          End
end
```

Como ves, hemos creado la variable *cont* y la hemos iniciado a 0, luego en el bucle la vamos decrementando, y cuando es negativa la volvemos a poner a cero. De esta manera seguirá a cero hasta que pulses “x”, que será cuando se cumplan las dos condiciones del último IF y se entre en él, allí se generará el disparo y se pondrá *cont* al valor de *retardo*, de forma que será necesario repetir el bucle “retardo” veces para volver a disparar. En caso de que no quieras que retardo sea de 10, te vas a la zona de constantes y pones el valor que te de la gana. Recuerda que para hacer una comparación se usa “==” y no “=” ya que lo segundo es una asignación (cambiar el valor de una variable).

Existen muchas otras maneras de evitar que el jugador dispare de forma infinita manteniendo la tecla pulsada y obligarle a tener que pulsar la tecla repetidas veces para disparar. Algunas ideas (trozos de código insertables en un proceso "nave()" o similar) que puedes probar en los juegos que puedas crear a partir de ahora podrían ser los siguientes:

```
if (key(_space)) //Se supone que la tecla SPACE es la de disparar
            if (bloqueo==0)
                  disparo(x,y);
                  bloqueo=1;
            end
else
            bloqueo=0;
end
```

Lo que se hace en este ejemplo es tener una variable con el estado anterior de la tecla, que vale 0 cuando no se pulsó y 1 cuando si se pulsó. Si se da el caso que antes no estaba pulsada (*bloqueo=0*) y ahora sí (*key(_space)*) crea un proceso de disparo, éste se encarga de actualizar la variable *bloqueo* a que está pulsada y no la pone como no pulsada hasta que desaparece el disparo.

Este otro ejemplo es similar al anterior, pero fuerza además a que mientras exista un disparo en la pantalla, no se pueda generar otro:

```
 // Este es el primero a comprobar... si este cumple nos ahorramos IFs
if (!exists(type disparo))
            if (key(_space))
                        if (!bloqueo)
                              disparo (x,y);
                              bloqueo=1;
                        end
            else
                        bloqueo = 0 ;
            end
end
```

Otra tercera manera, donde sólo puede existir un disparo en cada momento,sería así: hay que poner lo siguiente en el proceso "nave":

```
if (bloqueo==0 and key(_space)) disparo(); end
```

y luego escribir el proceso "disparo" así:

```
process disparo()
begin
  father.bloqueo=1; //Depende cómo hayas declarado bloqueo, aqui está como LOCAL
  repeat
  //... desplazas el disparo y eso
  until (y<150)
  father.bloqueo=0;
end
```

En este último ejemplo, si quisieras tener más de un disparo en la pantalla a la vez, tendrás que modificar el proceso "nave()" a algo así:

```
if (bloqueo==0 and key(_space)) disparo(); bloqueo=1; end
if (bloqueo==1 and !key(_space))bloqueo=0; end
```

y en el proceso "disparo()" eliminar las líneas que hacer referencia al father (la primera y la última).

## *Añadiendo explosiones

Para crear una explosión cuando un enemigo impacte con un disparo,vamos a hacer lo mismo que hicimos en el capítulo del juego del laberinto: una animación, es decir, vamos a crear la ilusión de movimiento y de cambio de imagen a través de imágenes fijas. Por supuesto, nuestro primer problema es conseguir los gráficos de una explosión para después meterlo en el FPG. Hay varias maneras de hacerlo, la primera es la más obvia, y es que cojas tu editor y te dediques a pintar el fuego expandiéndose y desapareciendo, algo seguramente difícil de hacer de forma realista. También podemos buscar una explosión en las numerosas librerías gratuitas que circulan por la red (o "ripear" los gráfico de otros juegos, cosa que, además de no ser legal, quita originalidad y personalidad a tu juego). Nosotros optaremos por la primera opción.

Crea 15 imágenes, cada una de ellas con una forma levemente diferente de lo que sería la explosión: las primeras imágenes pueden ser puntos rojos pequeños y las últimas pueden ser grandes llamaradas amarillas y azules. Usa tu imaginación. Finalmente, añade los gráficos creados al FPG en el orden cronológico de la explosión. En los ejemplos su código irá del 005 al 019.Bien, ya tenemos la base de la animación, ahora es el momento de ponernos manos a la obra e indicarle al ordenador lo que queremos hacer. Para empezar, esta animación sólo ocurrirá cuando el enemigo muera, por lo que será él el encargado de generarla ¿Crearemos un proceso nuevo "explosion()" como hicimos en el tutorial del laberinto? Pues ahora no: veremos otra solución alternativa. Lo que haremos será cambiar el gráfico del enemigo por la del fuego, simplemente. Esto lo haremos dentro del proceso "enemigo()", después de haber comprobado que un disparo ha impactado con él. Y la idea de generar la animación es creando un bucle, en el que irá cambiando la imagen e irá mostrándose una a una a cada frame de esta forma tendremos la animación.

Tenemos que hacer un bucle, y tenemos varias formas de hacerlo, pero las mejores son aquellos bucles que tienen definido un número determinado de repeticiones, pudiendo usar FROM y FOR, esto ya es cuestión de gustos. El que quizás sea más complejo sea el bucle FOR, pero es el que más posibilidades da, y se usa en muchos lenguajes, por ejemplo en C++, que es de los más usados, y así tendrás un ejemplo práctico de cómo se usa. Crearemos una variable *cont* como contador, y añadiremos todas las líneas marcadas en negrita:

```
Process enemigo (x,int inc_x,int inc_y)
Private
          Int ID_disparo_acertado;
          Int cont;
end
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    ID_Disparo_acertado=collision(type disparo);
                    If (ID_disparo_acertado !=0)
                              Signal(ID_disparo_acertado,s_kill);
                              break;
                    End
                    Frame;
          Until (y > 520)
          Cont=5;
          For(cont=5;cont<=19;cont++)
                    Graph=cont;
                    Frame;
          End
end
```

Poco más hay que explicar, cada frame muestra una imagen distinta, en orden, dando la impresión de animación. Ejecútalo y verás que efecto más fantástico. Haz pruebas con diversas explosiones de distintos tamaños, colores o número de imágenes hasta que des con lo que buscas.

## *Añadiendo energía enemiga

Ahora vamos a añadir algo tan interesante como es la energía para los enemigos, muy corriente para crear distintos tipos de enemigos con armaduras potentes, incluidos los pesados de los jefes finales. Nosotros vamos a crear algo muy básico aquí, todos los enemigos tendrán la misma energía, en caso de que quieras crear algo más complejo ya te daré alguna orientación.

Por lo pronto, nuestros enemigos podrán resistir tres de nuestros ataques (obviamente más si tú lo programas). Para ello necesitaremos una variable *energia*. ¿Pero ha de ser global, privada, pública...? Pensemos: si otro proceso fuera a consultar su energía no podríamos declararla como privada; tampoco global, pues necesitamos una por enemigo y no sabemos a ciencia cierta cuantos va a haber como máximo. Hacerla variable local serviría en caso de que otro proceso dependiera de si un enemigo concreto está muerto, o le queda poca energía,etc, pero recuerda que entonces la variable energía también estará presente en otros procesos como disparo, el principal o incluso la nave protagonista, aunque no la necesiten para nada. Podría ser pública, pero como en nuestro caso la variable *energia* no va a ser consultada para nada por ningún otro proceso, ya que nuestros enemigos aparecen, bajan y mueren sin ningún control, usaremos una variable privada.

Tras declararla la vamos a inicializar a 3, que será su energía inicial. Y el descenso de energía enemiga lo vamos a implementar así: una vez que impacte un disparo nuestro, en el código de "enemigo()" se entrará de nuevo en el IF que “mataba” el disparo y salía del bucle, pero ahora lo que vamos a escribir es que se reduzca la energía en una unidad, y en caso de que sea cero, entonces sí se salga del bucle y muera. Parece sencillo y lo es, sólo hay que ponerse y escribirlo:

```
Process enemigo (x,int inc_x,int inc_y)
Private
          Int ID_disparo_acertado;
          Int cont;
          Int energia=3;
end
Begin
          Graph=4;
          Size=rand(65,115);
          Y=-40;
          Repeat
                    X=X+inc_x;
                    Y=Y+inc_y;
                    ID_Disparo_acertado=collision(type disparo);
                    If (ID_disparo_acertado !=0)
                              Signal(ID_disparo_acertado,s_kill);
                              Energia=energia-1;
                              If (energia==0)
                                        Break;
                              End
                    End
                    Frame;
          Until (y > 520)
          Cont=5;
          For(cont=5;cont<=19;cont++)
                    Graph=cont;
                    Frame;
          End
end
```

## *Añadiendo energía nuestra y su correspondiente barra gráfica. Introducción al uso de regiones

Vamos a apañarnos ahora una barra de energía para nosotros, aunque de momento no vamos a tratar la muerte de nuestra nave . Lo primero que necesitamos es el gráfico de la barra de energía, esta tendrá 200 pixeles de

ancho, y el alto que quieras (yo propongo 20). Puedes decorarlo como gustes, aunque sería bonito que hagas un degradado de color (que pase de un color a otro de forma progresiva); puedes usar un color vivo, que resalte sobre el escenario y los enemigos. Métela en el FPG con código 006.

También necesitamos declarar la variable de la energía de nuestra nave, *energia_jugador*, así que de nuevo tenemos que plantearnos el tipo que será. Lo más lógico es que sea única, por lo que las posibilidades se reducen a dos: global o privada, ahora depende de cómo vamos a acceder a ella. Podemos controlarla perfectamente desde el proceso "nave()", o podemos crear un proceso aparte. Lo cierto es que da igual qué método usemos, es cuestión de gustos, no hay una norma fija, aunque sí es recomendable separar procesos que controlan cosas distintas, sobre todo para mantener la claridad a la hora de corregir o mostrar el programa a otra persona; aunque el motivo que nos hace decantar por una solución es otro: un gráfico necesita un proceso. Por eso crearemos uno nuevo, "energia_nave()" que muestre la energía. Y por tanto, la variable *energia_jugador* ,entonces, será global, porque va a ser utilizada por más de un proceso: como mínimo por el proceso "energia_nave()", el proceso "nave()" y –ya lo veremos- por el proceso "enemigo()".

Te estarás preguntando cómo vamos a conseguir que nuestra barra de energía se encoja y estire a voluntad según la energía que tenga nuestra nave. Si eres un poco avispado estarás pensando en cómo hacerlo con la variable ***SIZE*** ó ***SIZE_X*** o bien usando una imagen que la tape. Pero vamos a usar otro sistema, las regiones de pantalla. Las regiones son zonas de un tamaño definido dentro de la pantalla, que pueden tener un comportamiento diferente al resto de ella. Las regiones tienen múltiples utilidades (saber si un gráfico está fuera de una región -con la función **out_region()**- , definir ventanas para multijugador a pantalla partida -con la función **define_region()**,etc-) pero en este juego , la que más utilidad le vamos a dar es que podemos meter un gráfico dentro de una región, de tal manera que si éste se sale de ella, desaparece como si fuera el borde de la pantalla. Las regiones se estudiarán con más profundidad en el apartado correspondiente del capítulo siguiente, así que si no entiendes algo de lo que se explica aquí, te remito a ese lugar del manual.

Nosotros vamos a jugar con esto último, haremos una región y meteremos dentro el gráfico, y haremos que la región se estreche y se ensanche haciendo que se muestre la porción de energía que le queda a la nave (es decir, que muestre una parte más o menos grande del gráfico). Evidentemente, habrá que encontrar una correspondencia entre la energía y la anchura en píxeles de la pantalla del gráfico.

Empecemos pues con el código. Lo primero es declarar la variable para la energía de nuestra nave. Aunque técnicamente podríamos, no es aconsejable llamarla *energia* porque ya hay una variable privada con ese mismo nombre en el proceso "enemigo()", es la importancia de buscar nombres claros pero bien elegidos, así que defínela como *energia_jugador* e inicialízala a 100 como ya sabes. Ahora coge nuestro proceso nave y haz una llamada al nuevo proceso justo antes del LOOP. Sólo necesitamos un proceso, lo llamaremos "energia_nave()". No olvides que no debes usar letras con tildes y de poner los paréntesis tras la llamada, aunque no pongas parámetros, son necesarios.

Momento de crear el nuevo proceso.

```
Process energia_nave ()
Begin
          X=320;
          Y=30;
          Graph=6;
          Loop
                    Frame;
          End
end
```

Poco hay que decir de esto, ya sabes que lo único que hace es mostrar un gráfico en una posición determinada. Ahora tenemos que definir la región, y para ello la función clave es **define_region()**. Definiremos la región 1, en la posición 320-100=220 horizontal (mitad de la pantalla menos mitad del gráfico) y 30-10=20 vertical (posición vertical del gráfico menos la mitad del alto de este), con un ancho del tamaño *energia_jugador* y un alto de 20, o sea, el alto del gráfico. Para modificarlo en cada frame, (para actualizar su ancho a lo que valga *energia_jugador*), es necesario volver a definirla, por eso será lo primero que hagamos al entrar al bucle.

```
Process energia_nave ()
Begin
          X=320;
          Y=30;
          Graph=6;
```

```
        Loop
                Define_region(1,220,20,energia_jugador,20);
                Frame;
        End
end
```

Teóricamente, como *energia_jugador* vale 100, la energía que debería verse es exactamente la mitad (no olvidemos que el gráfico mide 200 de ancho). Sin embargo pasan dos cosas: ni la energía es la mitad, ni se ve por encima de los demás procesos. Lo segundo se arregla fácilmente asignando antes del bucle el valor –1 a la variable ***Z***, como vimos en el proceso disparo, salvo que ahora queremos que se muestre por encima.

¿Qué falla en cuanto a lo de la región? Tan sencillo como que no le hemos dicho al ordenador que el proceso sólo debe ser visible dentro de esa región que definimos. Para ello, tenemos que decir explícitamente que tal proceso o tal otro "pertenecen" a una región, y por tanto, sólo serán visibles dentro de ésta. Y eso lo hacemos mediante la variable local ***REGION*** del proceso en cuestión. Asignándole el número de la región, el proceso solamente se verá dentro de ésta.

```
Process energia_nave ()
Begin
        X=320;
        Y=30;
        Graph=6;
        Region=1;
        Loop
                Define_region(1,220,20,energia_jugador,20);
                Frame;
        End
end
```

Ahí lo tienes, justo la mitad. Haz la prueba variando el valor de *energía_jugador*. Por supuesto esto no lo podrás hacer durante el juego, hay que controlarlo según los eventos que sucedan en el juego.La energía puede bajar por los disparos enemigos o por chocar con los enemigos. No vamos a hacer que los enemigos puedan disparar porque crear una inteligencia artificial que lo permita no es algo que se haga fácilmente: podríamos hacer que aleatoriamente los enemigos generasen en cada frame un número entre 1 y 50 y que dispararan cuando sacaran 25 (o un número dentro de un rango) y lo hicieran hacia el jugador (es sencillo si se sabe usar las funciones **get_angle()** y **advance()**, hay ejemplos de ello en el próximo capítulo). Pero nosotros nos centraremos sólo en el choque con nuestra nave.

Dado que los enemigos chocarán con nosotros durante varios frames, haremos que nos quiten un poco de energía en cada uno de ellos. Si controlamos esto desde el proceso de nuestra nave, sólo se nos quitará energía como si chocásemos con una enemiga, aunque sean dos las que lo están haciendo a la vez, por eso lo controlaremos desde las naves enemigas, esta es una de las ventajas de haber definido *energia_jugador* como variable global. A escribir:

```
Process enemigo (x,int inc_x,int inc_y)
Private
        Int ID_disparo_acertado;
        Int cont;
        Int energia=3;
end
Begin
        Graph=4;
        Size=rand(65,115);
        Y=-40;
        Repeat
                X=X+inc_x;
                Y=Y+inc_y;
                ID_Disparo_acertado=collision(type disparo);
                If (ID_disparo_acertado !=0)
                        Signal(ID_disparo_acertado,s_kill);
                        Energia=energia-1;
                        If (energia==0)
                                Break;
                        End
                End
                If(collision (type nave))
                        Energia_jugador=energia_jugador-1;
```

```
                    End
                    Frame;
          Until (y > 520)
          Cont=5;
          For(cont=5;cont<=19;cont++)
                    Graph=cont;
                    Frame;
          End
end
```

Pero ¡cuidado! Si lo hacemos así llegará un momento en que *energia_jugador* podría ser negativa, esto hay que controlarlo. El proceso que controla la energía de nuestra nave es el proceso "energia_nave()", así que añadimos un IF:

```
Process energia_nave ()
Begin
          X=320;
          Y=30;
           Z=-1;
          Graph=6;
           Region=1;
          Loop
                    If (energia_jugador<0)
                              Energia_jugador=0;
                    End
                    Define_region(1,220,20,energia_jugador,20);
                    Frame;
          End
end
```

Fíjate dónde lo hemos puesto, no es casualidad, **define_region()** no admite números negativos, por eso lo hemos puesto antes, así nos aseguramos que siempre *energia_jugador* sea positivo.

Pero si siempre pierdes energía, el jugador se sentirá estafado y dejará de jugar por morir tan pronto, así que hay que suministrarle pilas. Podemos hacer que aparezcan procesos que al chocar con el jugador le suministre la ansiada energía, pero vamos a ser un poco más originales y haremos que la energía suba sola al matar a un enemigo. Es tan fácil como añadir la línea *energia_jugador=energia_jugador+3;* en el proceso "enemigo()" justo antes del *break;* que hay dentro del *If(energia==0)*. Podrías pensar que podríamos haberla escrito tras salir del bucle principal REPEAT/UNTIL, ya que el proceso enemigo está a punto de finalizar, pero piensa que también se aumentaría la energía por desaparecer por debajo de la pantalla.

A más a más,acuérdate de controlar que la energía de la nave no sobrepase el valor 200, porque ésta sigue creciendo y creciendo y la región definida puede salirse de la pantalla y dar un error que te cuelgue el juego. Así que añadiremos otro IF:

```
Process energia_nave ()
Begin
          X=320;
          Y=30;
           Z=-1;
          Graph=6;
           Region=1;
          Loop
                    If (energia_jugador<0)
                              Energia_jugador=0;
                    End
                    If (energia_jugador>200)
                              Energia_jugador=200;
                    End
                    Define_region(1,220,20,energia_jugador,20);
                    Frame;
          End
end
```

De momento no hemos hecho nada para cuando nuestra nave llega a energia_jugador=0. Podríamos hacer

que desapareciera de la pantalla y que se mostrara un bonito "GAME OVER". Para hacer esto, es fácil de ver que lo que habría que modificar es el bloque *if(energia_jugador<0)/end* del proceso “energia_nave()” de la manera siguiente:

```
If(energia_jugador<0)
      energia_jugador=0;
      write(0,320,240,1,"GAME OVER!!!");
//Hacemos desaparecer la nave.Por tanto ya no podemos jugar
      signal(Type nave,s_kill);
      Break;
End
```

## *Disparo mejorado y disparo más retardado

Ahora vamos a añadir un efecto nuevo al juego: vamos a hacer que si la nave protagonista supera un umbral mínimo de energía, la frecuencia de los disparos también disminuya. Es decir, que si tienes poca energía, tus disparos serán más lentos en generarse.Puede que no te guste la idea, darle desventaja al jugador cuando más apoyo necesita, pero es cosa de los juegos, sólo hay recompensa si juegas bien.

La modificación de esto se puede llevar a cabo en varios sitios: en nuestro nuevo proceso “energia_nave()”, en el proceso “nave()”... pero dado que nuestra nave reaccionará más adelante de otra forma –ya lo veremos-, lo haremos en su proceso. Lo que haremos será alterar el valor de *retardo*, aumentándolo, para que tarde más en disparar. A estas alturas es sencillo que lo hagas tú sólo: tienes que comprobar si la energía es menor de cierto valor y asignar el nuevo valor, y si no, restaurar el anterior. Pero vamos a usar un nuevo método: usaremos SWITCH. Aunque hay un detalle que se te puede haber pasado por alto: *retardo* no es una variable, es una constante, y por lo tanto no se puede modificar. Bueno, tan sencillo como cambiar esa línea a la zona de variables globales y listo. Y además, el proceso "nave()" quedaría así (hemos añadido el SWITCH):

```
Process nave ()
Private
          Int cont=0;
End
Begin
          Graph=2;
          X=320;
          Y=435;
          energia_nave();
          Loop
                    Cont=cont-1;
                    If(cont<0) cont=0; end
                    If (key(_left)) x=x-4; end
                    If (key(_right)) x=x+4; end
                    If (x>600) x=600; end
                    If (x<40) x=40; end
                    Switch(energia_jugador)
                              Case 0..30:
                                        Retardo=10;
                              End
                              Default:
                                        Retardo=5;
                              End
                    end
                    If (key(_x) AND cont==0)
                               cont=retardo;
                               disparo();
                    end
                    Frame;
          End
end
```

Si quieres puedes añadir distintos retardos en función de la energía. Procura no repetir valores, de lo contrario, el primer rango que cumpla la condición será el que se ejecute y se ignorará el resto.

Vamos ahora a darle una alegría al jugador, cuando tenga mucha, pero mucha energía lo que vamos a hacer es aumentar la capacidad de disparo, pero de una forma un tanto especial: los enemigos seguirán soportando tres impactos, pero dispararemos doble: saldrán dos disparos cada vez. Lo primero que tenemos que hacer es coger el proceso "disparo()" y hacer un ligero cambio, y es que en vez de recoger el valor de su variable ***X*** a partir de la línea *X=father.x;* vamos a conseguir lo mismo pero pasándole ese valor como parámetro, igual que hicimos en el tutorial del laberinto:

```
Process disparo (x)
Begin
          Graph=3;
          Y=410;
          Z=1;
          Repeat
                    y=y-15;
                    frame;
          until (y < -15)
end
```

Y además, modificaremos el código del proceso "nave()" de la siguiente manera:

```
Process nave ()
Private
          Int cont=0;
End
Begin
          Graph=2;
          X=320;
          Y=435;
          energia_nave();
          Loop
                    Cont=cont-1;
                    If(cont<0) cont=0; end
                    If (key(_left)) x=x-4; end
                    If (key(_right)) x=x+4; end
                    If (x>600) x=600; end
                    If (x<40) x=40; end
                    Switch(energia_jugador)
                              Case 0..30:
                                        Retardo=10;
                              End
                              Default:
                                        Retardo=5;
                              End
                    end
                    If (key(_x) AND cont==0)
                               cont=retardo;
                                If(energia_jugador > 195)
                                      disparo(x-10);
                                      disparo(x+10);
                                else
                                     disparo(x);
                                end
                    end
                    Frame;
          End
end
```

Lo que hemos hecho simplemente ha sido poner que si la energía del jugador es mayor de 195, que dispare dos disparos en vez de uno, separados 20 píxeles uno del otro.

## *Añadiendo los disparos de los enemigos

¿Qué sería de un juego de matamarcianos si éstos no te pueden disparar? Hagámoslo.Los disparos de los

enemigos se podrían hacer de muchas maneras: que cayeran siempre verticalmente hacia abajo, que persiguieran a nuestro protagonista allí donde se moviera…lo que está claro es que los disparos de los enemigos tendrán que ser otro proceso diferente del de los disparos nuestros: éstos hemos hecho que siempre fueran para arriba, ¿recuerdas? y este comportamiento precisamente no es el que queremos para los disparos enemigos.

Haremos el caso más simple:los disparos de los enemigos simplemente se moverán incrementando su Y. La otra alternativa propuesta, que los disparos fueran teledirigidos hacia nuestra nave, no es excesivamente difícil de hacer utilizando las funciones **get_angle()** y **advance()**, por ejemplo; ya lo hemos comentado unos párrafos antes

Así pues, creamos un nuevo proceso que representará el disparo enemigo, llamado "edisparo()":

```
Process edisparo(x,y)
Begin
/*El gráfico puede ser otro. Aquí utilizamos el mismo gráfico que el de los disparos
nuestros*/
           graph=3;
           Z=1;
           Repeat
                y=y+5;
                if(collision(TYPE nave))
                      energia_jugador=energia_jugador-1;
                end
                frame;
           until(y>480)
FRAME;
END
```

Y en el proceso "enemigo()", lo creamos. Escribe la siguiente línea en el interior del bloque REPEAT/UNTIL de dicho proceso, justo antes de la orden frame;.

```
if(rand(0,100)<10) edisparo(x,y); end
```

Y ya tendrás enemigos temibles: te podrán quitar energia al chocar contra ti y al acertar con sus disparos. ¡O sea que cuidado!

## *Añadiendo los puntos. Introducción al uso de las fuentes FNT

Lo que vamos a hacer ahora es mostrar los puntos que la nave irá sumando cada vez que haga explotar un enemigo (1 punto=1 enemigo muerto), en la parte superior de la pantalla, centrado respecto al ancho Lo primero es crear una variable *puntos* que contenga el valor de los puntos, declárala de tipo global en el programa principal, y a continuación vamos a poner la condición para que sume un punto: esto último lo haremos añadiendo la línea en el proceso enemigo, como podemos ver a continuación:

```
puntos=puntos+1;
```

justo antes del BREAK del interior de la sentencia *If(energia==0)* del proceso "enemigo()".

El escribir el texto lo vamos a hacer desde el proceso principal, aunque no es lo más recomendable si quisiéramos tener el código bien agrupado y ordenado; lo bueno sería crear un proceso nuevo que controlara los textos, pero como sólo vamos a escribir éste no hará falta. Y lo vamos a hacer bonito: vamos a utilizar una fuente FNT para mostrar el texto (el tema de las fuentes FNT también se trata en el capítulo siguiente: remito el lector al apartado correspondiente para obtener más información).

Resumiendo mucho, hay que hacer lo mismo que con las imágenes. Primero se carga la fuente, y para ello tenemos **load_fnt()**, donde el parámetro que hay que pasarle es la ruta absoluta o relativa –según convenga- del archivo de fuente, y al igual que pasa cuando se carga una imagen, también esta función devuelve un número identificador. Por ejemplo, tenemos un tipo de letra que hemos llamado "mi_letra.fnt" y hemos declarado una variable global para almacenar su ID llamada *letra_id*, la línea para cargarlo sería *letra_id=load_fnt("mi_letra.fnt");* siempre y cuando el archivo de fuente este en la misma carpeta desde donde se ejecuta el DCB.

En este momento podrás entender mejor cuál es el significado del primer parámetro que siempre valía 0 hasta ahora de la función **write()**. En su momento te dije que poner 0 ahí equivalía a utilizar la fuente predeterminada del sistema, la cual no hay que cargar. En realidad, este primer parámetro lo que indica es el identificador de la fuente con la que queremos escribir el texto. Así, si hemos cargado una fuente con **load_fnt()** y esta función nos devuelve un identificador que lo guardamos en una variable *letra_id*, si escribimos por ejemplo *write(letra_id,100,100,1,"Hola");*, aparecerá la palabra "Hola" escrita con la fuente identificada por *letra_id*, que será la que hemos cargado con **load_fnt()**.

En resumen, si queremos imprimir los puntos en pantalla –en este ejemplo usaremos la fuente del sistema: si quieres utilizar otra fuente ya sabes cómo hacerlo-, deberemos incluir la siguiente orden dentro del proceso principal, justo antes de su LOOP.

```
write_var(0,320,10,1,puntos);
```

Fíjate que usamos **write_var()** y no **write()** porque queremos que cada vez que la variable *puntos* cambie de valor, allí donde lo haga automáticamente, la impresión de ese valor también cambie.

Supongo que ya te habrás dado cuenta de que nuestro juego no tiene fin. Puedes estar matando enemigos toda la vida. Estaría bien, por ejemplo, que nuestro juego acabara cuando la nave hubiera eliminado un cierto número de enemigos, por ejemplo 100. En ese momento, podría aparecer sobreimpresionado algún texto felicitando al jugador, desapareciendo además todo rastro de presencia enemiga, para que nuestra nave pudiera viajar a sus anchas por todo el universo.Vamos a hacerlo.Es fácil ver que lo que tendríamos que hacer es añadir en el proceso "nave()", justo antes de su orden FRAME, el siguiente bloque IF/END:

```
If (puntos==100)
       signal(Type enemigo,s_kill);
       write(0,120,240,0,"¡Muy bien! ¡Has sorteado los asteroides con éxito!.");
End
```

## *Introducción al uso del ratón

Lo que vamos a hacer ahora va a ser comprobar la posición horizontal del ratón y colocar la nave en su posición, y no sólo eso, sino que vamos a usar los mismos botones del ratón para disparar. Otra vez recomiendo el lector la lectura previa del apartado correspondiente del capítulo siguiente para obtener información más completa y extensa sobre el uso del ratón. No obstante, para conseguir lo que queremos, es tan sencillo como cambiar un par de líneas:

```
Process nave ()
Private
          Int Cont=0;
End
Begin
          Graph=2;
          X=320;
          Y=435;
          Energia_nave();
          Loop
                    Cont=cont-1;
                    If (cont<0) cont=0; end
                    X=mouse.x;
                    Switch(energia_jugador)
                              Case 0..30:
                                        Retardo=10;
                              End
                              Default:
                                        Retardo=5;
                              End
                    end
                    If (mouse.left and (cont==0))
                                        Cont=retardo;
```

```
                                        If(energia_jugador>195)
                                                Disparo(x-10);
                                                Disparo(x+10);
                                        else
                                                disparo(x);
                                        end
                        end
                        Frame;
                end
end
```

Fíjate que hemos sustituido cuatro líneas que controlaban las teclas de desplazamiento lateral por una sola, y además el manejo es más sencillo para el jugador. Puedes intentar modificar este proceso para que también se desplace verticalmente, pero no olvides cambiar la posición inicial vertical del proceso "disparo()". También controlas el disparo con el ratón, incluso sigue manteniendo el retardo y únicamente cambiando una comprobación. Los usos del ratón son tan variados como tu imaginación quiera, ahora es cosa tuya seguir investigando y probando.

## *Introducción al uso de scrolls de fondo

Nuestro juego ya es muy, muy completo, no te quejarás: tu nave se mueve, dispara, tiene energía, aparecen enemigos, usamos el ratón. Pero es soso, tiene el fondo ahí, estático, sin vida. Puede que si le has puesto algún planeta tenga algo de vida, y si te has atrevido a que se muevan y todo pues todavía, pero es que permanecer sin moverse en esta pequeña parcela del espacio...Vamos a hacer que el fondo se mueva., simulando que nuestra nave está viajando y avanzando sin descanso a través del espacio sideral.

Para ello, vamos a implementar un "scroll automático". Automático quiere decir que consiste en un scroll que se mueve solo, sin depender de ningún otro proceso (normalmente el proceso protagonista, el cual va avanzando). Los scrolls automáticos suelen usarse para decorados de fondo con movimiento, como es nuestro caso: nuestra nave moviéndose por el espacio. El tema de los scrolls se trata ampliamente en el capítulo siguiente: si se desea profundizar en este tema y entender mejor los conceptos aquí tratados, recomiendo su lectura previa.

Para mostrar un fondo de scroll, lo que tenemos que hacer es modificar el programa principal de la siguiente manera: añadiendo las líneas de **start_scroll()** y **stop_scroll()** y quitando **put_screen()**:

```
Process Main()
Begin
        set_mode(640,480);
        set_fps(60,1);
        Graficos=load_fpg("prueba.fpg");
        Start_scroll(0,graficos,1,0,0,2);
        Nave();
        Write_var(0,320,10,1,puntos);
        Loop
                If (rand(0,100)<20)
                        Enemigo(rand(0,640),rand(-5,5),rand(4,7));
                End
                If (key(_esc)) break; end
                Frame;
        End
        Stop_scroll(0);
        Unload_fpg(graficos);
        Let_me_alone();
End
```

Vemos que creamos un scroll, el número 0, el cual estará formado por la imagen principal 001 del FPG referenciado por la variable *graficos* (es decir, el fondo estrellado), y que no tendrá imagen "de fondo" de scroll (los scrolls pueden tener dos imágenes: una más en primer plano y otra más de fondo, aunque ambas estarán por defecto a una profundidad mayor que todos los procesos, y de hecho, ambas imágenes tienen la misma ***Z*** entre sí). Como región definida tiene toda la ventana del juego, por lo que el scroll (si la imagen es lo suficientemente grande, que lo es) se producirá en toda la ventana; y como útlimo parámetro tiene el valor 2, cosa que quiere decir que si el scroll se mueve verticalmente, lo hará de forma indefinida (cíclica).

Es evidente que el scroll ha de acabar cuando el propio proceso principal finalice: por eso se ha escrito el **stop_scroll()** al salir del bucle, inmediatamente antes de la terminación del programa.

Quizás te hayas llevado una desilusión cuando hayas visto que el fondo no se ha movido, y es que, vale, lo has iniciado,(prueba de ello es que el fondo aun está ahí aún habiendo quitado el **put_screen()**), pero no le has dicho que se mueva. Para conseguir ese movimiento hay que modificar los campos de una estructura predefinida: ***SCROLL***.En concreto, los campos **"x0"** e **"y0"**, las cuales controlan la coordenada x e y del primer plano del scroll, respectivamente. Así que lo que nos falta por escribir, justo antes del frame del proceso principal es la línea :

```
scroll[0].y0=scroll[0].y0-2;
```

Lo que hemos hecho con esta nueva línea ha sido simplemente decirle al ordenador que en cada frame, la coordenada y de la imagen "en primer plano" –el 3r parámetro de **start_scroll()**- del scroll 0 disminuya en 2 píxeles, con lo que veremos que en apariencia la imagen del cielo estrellado se va moviendo hacia arriba. Y como hemos puesto el valor 2 en el último parámetro de **start_scroll()**, nos aseguramos que ese movimiento vertical sea cíclico.

Haz una última prueba: cambia el gráfico de fondo del scroll por el mismo que usamos para "el primer plano" del scroll, es decir, modifica la función por *start_scroll(0,graficos,1,1,0,2+8);* (no olvides también cambiar el último parámetro para que el movimiento del gráfico del fondo del scroll también sea cíclico). Ahora salen más estrellas porque estamos viendo dos imágenes del cielo estrellado: la de "primer plano", que se mueve, y la "del fondo" que acabamos de poner, que es la misma pero no se mueve. Si queremos que se mueva puedes usar los campos **"x1"** y **"y1"** de la estructura ***SCROLL***, así: *scroll[0].y1=scroll[0].y1-1;* :verás entonces que el espacio toma profundidad. Fíjate que hemos hecho que el incremento es diferente (en la imagen de "primer plano" cada frame la coordenada y disminuye en 2 píxeles y en la imagen "de fondo" disminuye en 1): esto es para crear un efecto de profundidad.

También puedes jugar con el campo **"ratio"** de la estructura scroll (*scroll[0].ratio*)si quieres. En ella se guarda la velocidad del fondo respecto al plano principal en porcentaje (así, si vale 100, el fondo se moverá a la misma velocidad que el plano principal, si vale 50 el fondo irá a la mitad de la velocidad, si vale 200 irá al doble de rápido y si vale 33 irá a un tercio,etc. Por lo tanto, en vez de espeficar de forma explícita los incrementos de las coordenadas x ó y para la imagen "de fondo", basta con hacerlo para la imagen "en primer plano" y establecer el campo "ratio".Así siempre tendrás que ambas velocidades de scroll están relacionadas una con la otra: si la del "primer plano" varía, la "del fondo" variará en consecuencia.

## *Introducción al uso de bandas sonoras y efectos de sonido

Remito al apartado correspondiente del capítulo siguiente, donde se explica y detalla la funcionalidad y uso de las funciones que Bennu aporta para la inclusión de músicas y efectos de sonido -para los disparos,explosiones,etc- en nuestros juegos (la familia de funciones **load_song(), play_song()** ,etc y **load_wav(), play_wav()**, etc, respectivamente).

# CAPITULO 8

## Órdenes y estructuras básicas de Bennu

### *Trabajar con la consola de depuración. Orden “Say()”

Los debuggers ( o depuradores) son un software específico utilizado por los programadores para buscar errores de varios tipos (léxicos, sintácticos,lógicos...) dentro de un código fuente. Son capaces de ejecutar el programa paso a paso, ejecutar sólo unas determinadas líneas repetidamente, inspeccionar variables, etc, en busca de dichos errores.

Depurar un programa sirve, pues, para detectar errores en tiempo de ejecución. Depurar un programa es como ejecutarlo “normal” pero con la posibilidad de parar su ejecución en cualquier momento que deseemos. Es entonces, al haber parado el juego, cuando tenemos acceso a los valores que en ese preciso momento tienen las diferentes variables del programa, información vital para detectar el posible error, ya que podremos observar en tiempo real cuándo dejan de valer aquello que se supone que deberían de valer, y poderlo corregir.

Para poder realizar este proceso de depuración, hay que hacer tres pasos:

**1)**Incluir en nuestro código fuente el módulo “mod_debug”.

**2)**Compilar nuestro código fuente escribiendo el parámetro “-g”. Es decir, así:

```
bgdc -g mijuego.prg
```

Con esta opción, el compilador almacenará en el archivo DCB información adicional, innecesaria para ejecutarlo (como los nombres de las variables o la posición de las instrucciones en los ficheros de los códigos fuente), pero que es imprescindible para poder depurar cuando ejecutemos el programa.

Si utilizas el entorno FlameBird, podrás hacer esto mismo si vas al menú “Edit”->”Preferences” y en la pestaña “Compilation” dejas marcada la opción “Compile in debug mode”.

**3)**Para depurar propiamente el programa, lo único que tendrás que hacer es ejecutarlo como siempre con *bgdi*, y en el momento que quieras, pulsar ***ALT+C*** . En ese momento, el juego se detendrá por completo (estará "congelado") y aparecerá una ventana que representa la consola desde la que podrás utilizar diferentes comandos especiales que te servirán de ayuda en la búsqueda del error díscolo: existen comandos para comprobar los valores que tienen las variables en ese mismo instante, también para mandar señales concretas a procesos concretos y observar lo que ocurre,etc,etc, ahora lo iremos viendo.
Para salir de la consola y continuar con el juego, basta con volver a pulsar ***ALT+C***, o bien ***F5*** o bien escribir el comando 'GO'.

Lo primero que haremos será consultar la ayuda de la consola para conocer qué comandos de depuración nos ofrece. Podemos escribir la palabra HELP o bien pulsar ***F1***. Nos aparecerá un listado con las órdenes más importantes, pero antes de nada, aprenderemos a movernos por la consola de debug. Por ejemplo, podemos deslizar hacia arriba, abajo, izquierda o derecha el contenido del listado de ayuda o de cualquier otro que obtengamos (ya que pueden ser más largos que el tamaño de la propia ventana de depuración) con la combinación de teclas ***CTRL+<Flechas de dirección>***. Para avanzar el contenido de dicha ventana podemos utilizar la tecla ***AvPag*** y para retroceder, la tecla ***RePag***, aunque si nos interesa, también podemos hacer el tamaño de la ventana de depuración más grande (o pequeña) con ***ALT+<Flechas de dirección>***. También nos será útil saber que con las flechas de dirección “arriba” y “abajo” accedemos al historial de comandos de debug, con lo podremos acceder a los comandos escritos anteriormente y volverlos a ejecutar sin tener que escribirlos. Finalmente, pulsando la tecla ***ESC*** borraremos lo que estemos escribiendo en ese momento en la consola

Una vez ya familiarizados con el entorno de la consola de depuración, podemos empezar a sacarle partido. Por ejemplo, podemos comprobar qué valores en el momento de la "congelación" del programa tienen las variables que deseemos. Para ello disponemos de una serie de comandos de consola que nos permitirán obtener esta información.

| | |
|---|---|
| **GLOBALS** | Muestra un listado del tipo y nombre de todas las variables globales del programa (incluyendo las variables globales predefinidas de Bennu), junto con sus valores actuales |
| **LOCALS nombreProceso** | Muestra un listado del tipo y nombre de todas las variables locales del programa (incluyendo las variables locales predefinidas de Bennu), junto con sus valores actuales. Si se escribiera un proceso del cual no existe en ese momento como mínimo una instancia que esté funcionando -activa, dormida o congelada-, sólo saldrá el listado de las variables locales predefinidas de Bennu sin ningún valor asociado. |
| **PUBLICS nombreProceso** | Muestra un listado del tipo y nombre de todas las variables públicas del proceso especificado, junto con sus valores actuales,siempre que exista como mínimo una instancia de ese proceso funcionando (activa, dormida o congelada) en este momento. |
| **PRIVATES nombreProceso** | Muestra un listado del tipo y nombre de todas las variables privadas del proceso especificado, junto con sus valores actuales,siempre que exista como mínimo una instancia de ese proceso funcionando (activa, dormida o congelada) en este momento. |
| **nombreProceso.nombreVariable** | Muestra el valor de la variable privada/pública/local especificada para un proceso dado. El programa principal es considerado como un proceso más, llamado "Main". No es posible mostrar el valor de una variable global concreta:para ello hay que recurrir al comando GLOBALS general. |

Vamos a practicar esto último. Para ello haremos servir un código bastante simple como el siguiente. Lo compilamos en modo depuración, lo ejecutamos y antes de que "miproceso()" acabe de escribir los números en pantalla, pulsamos ***ALT+C***:

```
Import "mod_text";
Import "mod_key";
Import "mod_debug";

global
  int miglobal;
end
process main()
begin
  miproceso();
  miglobal=65; //Cambio el valor inicial por defecto de una variable global por otro
  repeat
          frame;
  until(key(_esc))
end

process miproceso()
private
   int i;
end
begin
x=400; //Cambio el valor por defecto de una variable local de "miproceso()" por otro
while(i<40)
          write(0,100,10+5*i,4,i);
          i++;
          frame;
end
end
```

Puedes probar de observar los valores actuales de las variables globales del programa (ya sean las predefinidas o bien

nuestra *miglobal*, que aparece al final del listado) con el comando 'GLOBALS'. También puedes observar los valores actuales de las variables privadas de "miproceso()" con 'PRIVATES miproceso'. O más específicamente, puedes obtener el valor de la variable *i* de esta manera 'miproceso.i'.

También tienes la posibilidad de cambiar dentro de la consola los valores actuales de la variable privada/pública/local que desees, para el proceso que quieras, de esta manera:

***nombreProceso.nombreVariable = valor***

Los valores numéricos pueden ser reconocidos como hexadecimales si su último dígito es el carácter "h", binarios si su último dígito es el carácter "b" o decimales (sin ningún carácter final específico). Los valores de cadena se pueden especificar tanto entre comillas simples como dobles. Prueba por ejemplo, mientras tienes la consola en marcha, de asignar un nuevo valor a la variable *i* de "miproceso()" (es decir, escribe 'miproceso.i = 30', por ejemplo. Verás que, si ahora cierras la consola y continúa la ejecución del programa -pulsa ***ALT+C*** otra vez, recuerda-, se producirá un salto en los números visualizados en pantalla, ya que se ha alterado el valor que iba teniendo la variable *miprivada* dentro del bucle.

En el caso de asignar un nuevo valor a una variable global, simplemente se haría así: ***nombreVariable=valor***

Existen otro conjunto de comandos de la consola de depuración relacionados con la manipulación de procesos:

| | |
|---|---|
| **INSTANCES** | Lista los identificadores de proceso de todas las instancias existentes en el instante actual (ya estén activas -con la marca "[R]",dormidas -con la marca "[S]"-, congeladas -con la marca "[F]"-, muertas -con la marca "[K]"- o a punto de morir mientras ejecutan su sección "OnExit" -con la marca "[D]"- ),además de su correspondiente nombre de proceso. Un atajo a este comando es la tecla ***F2*** |
| **RUN nombreProceso [params opcionales]** | Ejecuta una nueva instancia del proceso especificado |
| **KILL identificadorProceso** | Elimina una instancia concreta del proceso especificado |
| **WAKEUP identificadorProceso** | Despierta una instancia concreta del proceso especificado |
| **SLEEP identificadorProceso** | Duerme una instancia concreta del proceso especificado |
| **FREEZE identificadorProceso** | Congela una instancia concreta del proceso especificado |
| **KILLALL nombreProceso** | Elimina todas las instancias del proceso especificado |
| **WAKEUPALL nombreProceso** | Despierta todas las instancias del proceso especificado |
| **SLEEPALL nombreProceso** | Duerme todas las instancias del proceso especificado |
| **FREEZEALL nombreProceso** | Congela todas las instancias del proceso especificado |

Prueba de abrir la consola una vez "miproceso()" haya acabado su ejecución (es decir, cuando hayan acabado de imprimirse los números en pantalla). En ese momento, escribe el comando 'INSTANCES'. Verás que sólo está funcionando el programa principal (proceso "main()") con un determinado código identificador. Si ahora ejecutas 'RUN miproceso', lo que estarás haciendo es que cuando salgas de la consola de depuración, se ejecute otra vez desde el principio el código del "miproceso()". Es posible que no te des cuenta que se está volviendo a ejecutar ya que imprimirá los números encima de los ya existentes en la pantalla, pero esto lo puedes cambiar facilmente por ejemplo haciendo que el segundo parámetro del **write()** sea un valor aleatorio y no 100: de esta manera, en cada ejecución de "miproceso()" los números aparecerán en posiciones horizontales diferentes (es una idea). De todas maneras, siempre podrás volver a la consola en esta nueva ejecución de "miproceso()" y comprobar que efectivamente, está funcionando si consultas otra vez lo que nos devuelve 'INSTANCES'.

De igual manera, puedes utilizar los demás comandos que envían señales a procesos concretos o a familias enteras, para que, al salir de la consola, éstas sean efectivas y se pueda comprobar su efecto en ese instante concreto. Por ejemplo, si añades justo antes del bucle LOOP/END del proceso principal una nueva línea que sea "miproceso();", estarás creando dos instancias de este proceso. Si ejecutas el programa y entras en la consola, haciendo 'INSTANCES' verás que aparecen efectivamente dos instancias de "miproceso()" con diferentes identificadores. Si ahora escribes por

ejemplo 'SLEEP idProceso', cuando salgas de la consola habrás dormido la instancia que hayas decidido (y por tanto, esa instancia no seguirá imprimiendo números en pantalla: lo podrás ver mejor si usar una coordenada horizontal aleatoria para el **write()**). Cuando ya se han acabado de imprimir los números correspondientes a la instancia que no se ha dormido, vuelves a entrar en la consola otra vez, podrás ver mediantes 'INSTANCES' que efectivamente, existe todavía una instancia de "miproceso()" dormida,y podrás despertarla usando 'WAKEUP idProceso'. Cuando salgas de la consola de nuevo, esa instancia reemprenderá su ejecución y acabará de mostrar los números en pantalla que le quedaban.

Otro ejemplo sencillo: si mientras están ejecutándose ambas instancias de "miproceso()", entras en la consola y escribes 'KILLALL miproceso', cuando salgas de ella se dejarán de escribir los números en la pantalla, ya que todas las instancias de "miproceso()" habrán muerto.

Hablando de otra cuestión: es posible que estés pensando que no es una manera muy profesional entrar en la consola durante la ejecución del juego pulsando ***ALT+C*** cuando se nos ocurra, porque la mayoría de las veces nos interesará entrar en la consola en un fotograma preciso del programa, para estudiar en ese instante concretísimo el estado de las variables y de los procesos. Con ***ALT+C*** no sabemos en realidad en qué momento preciso de la ejecución estamos, ya que depende de cuando hayamos pulsado estas teclas estaremos en una línea o en otra, hecho totalmente inaceptable si queremos realizar una depuración exhaustiva y precisa. Pues bien, la manera de entrar en la consola de depuración llegado un punto concreto de nuestro programa es mediante el uso del comando DEBUG; (sin paréntesis) de Bennu, el cual es una palabra reservada. Escribiendo este comando en cualquier sitio de nuestro código, la consola de depuración puede será invocada inmediatamente (sin esperar a ningún frame) cuando se llegue a ese sitio. A partir de entonces, tendremos disponible la consola justo en el momento que la necesitamos.

Puedes probar de insertar la orden DEBUG; en cualquier puntodel código del ejemplo, para comprobar su efecto. Por ejemplo, si escribimos lo siguiente dentro del bucle WHILE/END de "miproceso()":

```
if(i==14) debug; end
```

observa lo que pasa. Se abre automáticamente la consola en el momento que nosotros hemos dicho en el código (de hecho, sale impresa la línea que la ha invocado dentro de éste). Puedes comprobar que esto es así si haces por ejemplo un 'PRIVATES miproceso' y compruebas que efectivamente *i* vale 14.

Una vez que has entrado en la consola de debug, es posible que quieras ejecutar el código fuente línea a línea individualmente,a partir del punto del juego en el que éste se ha quedado congelado, para ir, por ejemplo, rastreando a cada línea los valores que van adquiriendo determinadas variables. Para lograr esto, dentro de la consola dispones del comando **TRACE**. Este comando, cada vez que se escriba, ejecutará una única línea del proceso donde actualmente se ha parado el programa. (mostrando el número de línea y además la propia línea en la consola, para saber cuál es), y volverá el control a la consola, para poder seguir utilizándola en este nuevo punto del programa. Un atajo para este comando es la tecla ***F8.***

Recuerda que la ejecución de los códigos de los procesos es consecutiva, uno tras otro (hasta que todos llegan a su frame y se visualizan en pantalla), así que si se utiliza varias veces seguidas la orden **TRACE;**, cuando se acabe de ejecutar las líneas de un proceso, se continuará por las líneas del siguiente. No obstante, si quieres pasar directamente a ejecutar el código del siguiente proceso "en la cola", puedes hacerlo con el comando **NEXTPROC** o bien pulsando la tecla ***F11***. Verás el proceso, el número de línea dentro de él y la propia línea de código a la cual has ido a parar.

Otro comando interesante es **NEXTFRAME**, el cual continúa la ejecución de todos los procesos hasta el siguiente frame, y retorna a la consola de debug, con lo que podremos observar de nuevo el estado de nuestro programa después del pase de un fotograma completo en todos los procesos.Un atajo para este comando es la tecla ***F10.***

Otro aspecto que se puede manipular con la consola de debug es la gestión de breakpoints. Un breakpoint es una línea determinada del código fuente en donde la ejecución se para cada vez que se llega ahí, para poder examinar en ese punto el estado del programa (variables, procesos,etc). Éste es un tema relativamente avanzado en el cual no profundizaremos, pero resulta una herramienta clave para el programador experimentado.

| | |
|---|---|
| **BREAK** | Lista los breakpoints definidos. |
| **BREAK nombreProceso** | Agrega un breakpoint en la ejecución del proceso especificado (se pueden también |

| | |
|---|---|
| | especificar funciones e instancias) |
| **BREAKALL** | Agrega un breakpoint a todas las instancias de todos los procesos |
| **BREAKALLTYPES** | Agrega un breakpoint a todos los tipos de procesos |
| **DELETE nombreProceso** | Elimina el breakpoint creado con "BREAK nombreProceso". Los breakpoints sobre una instancia determinada se eliminan cuando ésta muere |
| **DELETEALL** | Elimina todos los breakpoints de todas las instancias de todos los procesos |
| **DELETEALLTYPES** | Elimina todos los breakpoints de todos los tipos de procesos |

La mayoría de los comandos comentados hasta ahora también se pueden ejecutar de otra manera. Si con la consola de depuración monstrándose, pulsar la tecla MAYUS, verás que aparece un cuadro en la esquina superior derecha indicándote los tipos de procesos que existen en el programa en cuestión (en nuestro caso, el proceso "Main()"y el proceso "miproceso()"). Y verás también que el menú de abajo ha cambiado para dar paso a otros atajos de teclado.
Si seleccionamos un determinado tipo de proceso del cuadro superior, y siempre que mantengamos pulsada la tecla MAYUS, con ***F2*** observaremos un breve resumen del estado de las instancias de ese proceso (si están ejecutándose, dormidas,etc), con sus respectivos identificadores y los de sus padres. Si pulsamos ***F9*** estaremos agregando un breakpoint en la ejecución del tipo de proceso seleccionado. Si pulsamos ***F6*** entraremos en otro menú relacionado con las instancias concretas del tipo de proceso seleccionado (el menú anterior está relacionado con los tipos de procesos en general), donde con ***F3*** podremos observar los valores actuales de las variables locales de la instancia seleccionada en el cuadro superior, con ***F4*** podremos observar los valores actuales de las variables privadas de la instancia seleccionada, con ***F5*** lo mismo pero para las públicas, con ***F9*** estaremos agregando un breakpoint en la ejecución de esa instancia concreta y con ***F6*** volvemos al menú anterior (el de las opciones para los tipos de procesos). Para volver al menú original de la consola, basta con soltar la tecla MAYUS.

Finalmente, otros comandos/variables interesantes de la consola de depuración son:

| | |
|---|---|
| **QUIT** | Sale de la consola de depuración y termina la ejecución del programa |
| **SHOW expresion** | Evalúa una expresión aritmética y devuelve el resultado. Por ejemplo "SHOW 4*5" |
| **VARS** | Muestra un listado con el nombre y valor de las variables internas de la consola (FILES, SHOW_COLOR,etc) |
| **FILES** | Variable interna de la consola que muestra la cantidad de archivos abiertos (en uso) |
| **SHOW_COLOR** | Variable interna de la cuyo valor permitido es cualquier color RGB |
| **STRINGS** | Muestra todas las cadenas existentes en memoria |

Dejando de lado por completo la consola de depuración, Bennu aporta un comando especialmente útil a la hora de detectar errores en nuestros programas: el comando **say()**, definido en el módulo "mod_say", el cual sólo tiene un parámetro que puede ser cualquier texto ó bien el nombre de variable que queramos.

Por ejemplo, con la línea *say("texto");* mostraremos ese texto que hemos escrito, y con la línea *say(mivar);* mostraremos el valor de la variable *mivar*...pero ¿dónde?. En la ventana del intérprete de comandos que se utilizó para lanzar, mediante *bgdi* ,nuestro programa. Puedes probarla si escribes la línea *say(i);* en el código fuente de ejemplo en sustitución del **write().** Verás que ahora los números aparecen en la ventana del intérprete de comandos que has utilizado para ejecutar el código (siempre, claro está, que hayas invocado a *bgdi* de esta manera).

¿Qué aporta de nuevo este comando respecto por ejemplo **write()**? Este comando es mucho más ligero, cómodo y facilita mucho la observación rápida de valores de variables (o lo que queramos) en tiempo de ejecución. La principal utilidad de esta función es escribir "chivatos", pequeñas inserciones de código que muestran en momentos clave el estado de variables importantes, o simplemente informan acerca de la ejecución de un trozo de código. Esto permite obtener información útil sobre el desarrollo del programa, de cara a detectar la causa de bugs molestos.

Existe otra función similar, definida en el mismo módulo "mod_say", llamada **say_fast()**,la cual admite un parámetro del mismo tipo: un texto o el nombre de una variable.

## *Trabajar con cadenas

Bennu incorpora bastantes funciones que pueden ayudar al programador en el tratamiento de cadenas de texto.Veamos unas cuantas:

*Nota: Las funciones **chr()**, **asc()** y **substr()** también son funciones "de cadena", pero están explicadas posteriormente en este capítulo, en el apartado de introducción de caracteres por teclado.*

### LEN("CADENA")

Esta función, definida en el módulo "mod_string", devuelve el número de caracteres que tiene la cadena pasada como parámetro (es decir, su longitud).

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : INT: Número de caracteres de la cadena dada

### LCASE("CADENA")

Esta función, definida en el módulo "mod_string", devuelve una cadena idéntica a la original, pero convirtiendo todas las letras mayúsculas a minúsculas (incluyendo letras con acentos o tildes).

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : STRING : Cadena resultante

### UCASE("CADENA")

Esta función, definida en el módulo "mod_string", devuelve una cadena idéntica a la original, pero convirtiendo todas las letras minúsculas a mayúsculas (incluyendo letras con acentos o tildes).

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : STRING : Cadena resultante

### STRCASECMP("CADENA","CADENA2")

Esta función, definida en el módulo "mod_string", realiza una comparación entre dos cadenas (sin distinguir las mayúsculas de las minúsculas), y devuelve un valor que indica su diferenciación:

-0 quiere decir que las cadenas son iguales.
-Un valor negativo indica que el primer carácter diferente entre las dos cadenas va después (por orden alfabético) en la primera cadena que en la segunda.
-Un valor positivo indica que el primer carácter diferente entre las dos cadenas va después (por orden alfabético) en la segunda cadena que en la primera.

PARÁMETROS :

STRING CADENA : Cadena
STRING CADENA2 : Otra cadena

VALOR DE RETORNO : INT : 0 si las cadenas son equivalentes

### STRREV("CADENA")

Esta función, definida en el módulo "mod_string", devuelve una cadena creada tras dar la vuelta a la cadena que

recibe como parámetro. Es decir, **strrev("ABCD")** devolvería "DCBA".

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : STRING : Cadena resultante

**LPAD("CADENA",TAMAÑO)**

Esta función, definida en el módulo "mod_string", devuelve una cadena creada tras añadir espacios (carácter ASCII 32: " ") al comienzo a la cadena original, hasta conseguir que alcance una longitud igual a la especificada como segundo parámetro. Si la cadena original tiene un tamaño igual o superior al especificado, esta función devuelve la cadena original sin ningún cambio.

También existe la función **rpad()**, que expande una cadena añadiendo espacios a la derecha, hasta el tamaño dado. Funciona igual que **lpad()**.

PARÁMETROS:

STRING CADENA : Cadena original
INT TAMAÑO : Tamaño deseado

VALOR DE RETORNO : STRING : Cadena resultante

**TRIM("CADENA")**

Esta función, definida en el módulo "mod_string",devuelve una cadena idéntica a la original, pero eliminando cualquier espacio o serie de espacios que encuentre al principio o al final de la misma. Por espacio se entiene cualquier caracter de código ASCII 9 (tabulador), 10 (salto de línea), 13 (retorno de carro) ó 32 (espacio).

PARÁMETROS: STRING CADENA : Cadena original

VALOR DE RETORNO: STRING : Cadena resultante

Un ejemplo de estas últimas funciones podría ser:

```
import "mod_text";
import "mod_key";
import "mod_string";

process main()
private
   string a="hola ";
   string b="adiós";
end
begin
   write(0,50,10,4,TRIM(a));
   write(0,50,20,4,LPAD(a,17));
   write(0,50,30,4,RPAD(a,17));
   write(0,50,40,4,UCASE(a));
   write(0,50,50,4,STRREV(a));
   write(0,50,60,4,STRCASECMP(a,b));
   write(0,50,70,4,LEN(a));
   write(0,50,80,4,ASC("1")); //Vista más adelante
   write(0,50,90,4,CHR(49));  //Vista más adelante
   write(0,50,100,4,SUBSTR(b,2,3)); //Vista más adelante
repeat
          frame;
until(key(_esc))
end
```

A continuación comento las funciones de conversión:

**ATOI("CADENA")**

Esta función, definida en el módulo "mod_string", convierte una cadena en un número entero. Dada una cadena que contenga una serie de dígitos, devuelve el número entero que representan.

Si la cadena contiene caracteres adicionales después de los dígitos, estos serán ignorados. Si la cadena comienza por caracteres que no corresponden a dígitos, la función no detectará el número y devolverá 0.

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO: INT : Número resultante

**ITOA(NUM_ENTERO)**

Esta función, definida en el módulo "mod_string", convierte un número entero en cadena. Dado un número entero, devuelve una cadena con su representación, sin separador de miles.

PARÁMETROS: INT NUM_ENTERO : Número entero a convertir

VALOR DE RETORNO : STRING : Cadena resultante

**FORMAT(VALOR, DECIMALES)**

Esta función, definida en el módulo "mod_string", convierte un número decimal a cadena, con separador de miles y con el número de decimales indicado.

Si no se especifica el segundo parámetro, la función usará un número adecuado de decimales para representar el número (que pueden ser 0, por lo que se devolverá la cadena sin punto decimal).

PARÁMETROS:

FLOAT VALOR : Valor a convertir
INT DECIMALES : *(opcional)* Número de decimales

VALOR DE RETORNO: STRING : Cadena resultante

Además de estas funciones descritas, también existen las funciones **atof()** y **ftoa()**, que trabajan igual que **atoi()** y **itoa()** respectivamente, pero usando números en coma flotante (decimales)

Veamos un pequeño ejemplo de estas funciones:

```
import "mod_text";
import "mod_key";
import "mod_string";
import "mod_video";

process main()
private
          int entera=4;
          float decimal=9.82;
          string cadena="123Hola";
          string cadena2="12.3Hola";
end
begin
   set_mode(500,400);
   repeat
          delete_text(0);
          write(0,250,50,4,"Conversión de un entero en cadena: "+ itoa(entera));
         write(0,250,80,4,"Conversión de una cadena a entera: "+atoi(cadena));
         write(0,250,110,4,"Conversión de un número decimal en cadena: "+ftoa(decimal));
         write(0,250,140,4,"Conversión de un número decimal a cadena con determinados
decimales: "+format(decimal,4));
```

```
          write(0,250,170,4,"Conversión de una cadena a número decimal: "+atof(cadena2));
          frame;
   until(key(_esc))
end
```

Y por último, comento las funciones de búsqueda:

**FIND("CADENA","BUSCAR",POSICION)**

Esta función, definida en el módulo "mod_string", busca la cadena especificada como segundo parámetro en el interior de la cadena especificada como primer parámetro, y devuelve la posición de la primera coincidencia que encuentre, o -1 para indicar que no encontró ninguna. Esta posición es el número de carácter desde la izquierda de la cadena: 0 para el primer carácter, 1 para el segundo, y así sucesivamente. El tercer parámetro, opcional, indica el primer carácter donde dará comienzo la búsqueda. La parte de la cadena a la izquierda de éste, será ignorada. Si no se especifica el parámetro, se empezará por el primer carácter (posición 0).

La búsqueda se hace de izquierda a derecha. Sin embargo, es posible hacer también la búsqueda de derecha a izquierda si se especifica como tercer parámetro un número negativo, que indicará además la primera posición a tener en cuenta (-1 para el último carácter de la cadena, -2 para el penúltimo, y así sucesivamente).

Es importante destacar que **find()** considera caracteres diferentes las minúsculas de las mayúsculas

PARÁMETROS:

STRING CADENA : Cadena original
STRING BUSCAR : Cadena a buscar
INT POSICIÓN : *(opcional)* Posición inicial de la búsqueda

VALOR DE RETORNO: INT : Posición del primer carácter de la primera aparición de la cadena buscada, o -1 si no se encontró

Un ejemplo de su uso sería:

```
import "mod_text";
import "mod_key";
import "mod_string";

process main()
private
  string Txt="¡Hola Mundo, hola!";
end
begin
  write(0,10,30,3,"Texto = ¡Hola Mundo, hola!");
  write(0,10,40,3,"FIND(H) = "+itoa(find(Txt,"H")));
  write(0,10,50,3,"FIND(Hola) = "+itoa(find(Txt,"Hola")));
  write(0,10,60,3,"FIND(Hola a partir del 5 caracter) = "+itoa(find(Txt,"Hola",4)));
  write(0,10,70,3,"FIND(M) = "+itoa(find(Txt,"M")));
  write(0,10,80,3,"FIND(Mundo a partir del penúltimo caracter) =
"+itoa(find(Txt,"Mundo",-2)));
  write(0,10,90,3,"FIND(!) = "+itoa(find(Txt,"!")));
  write(0,10,100,3,"FIND(A) = "+itoa(find(Txt,"A")));
  write(0,160,190,4,"Pulsa ESC para Salir...");
  repeat
    frame;
  until(key(_esc))
end
```

**REGEX("EXPRESION","CADENA")**

Esta función, definida en el módulo "mod_regex", busca una expresión regular dentro de una cadena. Es decir, dada una expresión regular, comprueba si la cadena dada la cumple en algún punto, y devuelve la posición (número de carácter desde la izquierda, empezando en 0) donde la encuentra por primera vez, o -1 si no se cumple.

Una expresión regular puede entenderse como una forma avanzada de búsqueda de cadenas. La expresión regular puede ser una serie de caracteres corrientes como "abc", con lo que esta función se comportará igual que la función **find()** haciendo una búsqueda. Pero la verdadera potencia es que la expresión regular puede contener toda una serie de caracteres "comodín" (metacaracteres)que tienen significados específicos.

Los siguientes metacaracteres son reconocidos:

| | |
|---|---|
| **"."** | Un punto quiere decir "cualquier carácter".<br>Por ejemplo, "ab.de" permite encontrar "abcde" y "abJde", pero no "abde". |
| **"^"** | El exponencial quiere decir "principio de cadena" si la expresión regular empieza por ^.<br>Por ejemplo, "^abc" permite encontrar "abcdef" pero no "fabcdef". |
| **"$"** | El dólar quiere decir "final de cadena" si la expresión regular acaba en $.<br>Por ejemplo, "abc$" permite encontrar "deabc" pero no "abcde". |
| **"[]"** | Los corchetes quieren decir "cualquier carácter de esta lista", y en su interior puede haber una serie de caracteres corrientes, o bien rangos de caracteres.<br><br>Por ejemplo "[abc]" permite encontrar "a", "b" ó "c", mientras que "[a-z]" permite encontrar cualquier letra entre la "a" y la "z", inclusives.<br><br>Los rangos pueden ser inacabados: "[- ]" permite encontrar cualquier carácter hasta el espacio, mientras "[ -]" permite encontrar cualquier carácter del espacio en adelante.<br><br>Además, se reconocen "clases de caracteres" POSIX, que son secuencias predefinidas de caracteres que se escriben con la sintaxis "[:clase:]" y pueden ser cualquiera de las siguientes:<br><br>[:alpha:] para cualquier letra<br>[:upper:] para letras mayúsculas<br>[:lower:] para minúsculas<br>[:alnum:] para letras o números<br>[:digit:] para números<br>[:space:] para espacios. |
| **"()"** | Los paréntesis permiten agrupar secuencias de caracteres y metacaracteres. Esto tiene varias utilidades: por ejemplo, se pueden usar los metacaracteres de repetición como "*" o "?" después del paréntesis de cierre;o se pueden poner varias secuencias dentro de los paréntesis, separadas por el signo **"\|"**. La expresión encontrará cualquiera de las posibilidades. Por ejemplo, "(abc\|efg)" encontrará "abcde" al igual que "defgh",pero no "bcdef" |
| **"*"** | El asterisco indica "0 o más repeticiones de lo anterior", donde "lo anterior" se refiere al último carácter o metacarácter. Este metacarácter, al igual que los metacaracteres de repetición que detallamos a continuación, es más útil concatenado detrás de secuencias con corchetes o paréntesis.<br><br>Por ejemplo, "[abc]*" encuentra cualquier secuencia de 0 o más caracteres "a", "b", o "c". Es decir, "[abc]*" permite encontrar "aabc", "bbbcca", "bca" o "a". De la misma forma, "(abc)*" permite encontrar "abcabcabc", "abcabc" o "abc", pero no "aabbcc".<br><br>Es importante tener en cuenta que 0 repeticiones también son válidas, por lo que la expresión regular "[abc]*" en realidad se cumple siempre, ya que cualquier cadena contiene 0 o más repeticiones de cualquier secuencia. |
| **"+"** | El signo + encuentra 1 o más repeticiones de lo anterior, del último carácter o metacaracter anterior a este símbolo. Por ejemplo, "[abc]+" encuentra "aabbcc" y "bccba", pero no "defg". |
| **"?"** | El interrogante encuentra 0 ó 1 repeticiones de lo anterior. Es decir, permite marcar un trozo de la expresión como opcional. |
| **"{}"** | Las llaves establece un rango de repeticiones de lo anterior. El rango son dos dígitos separados por comas, aunque uno de ellos puede omitirse.<br><br>Por ejemplo, "[abc]{2,}" encuentra cualquier secuencia de dos o más caracteres "a", "b" o "c", mientras "[abc]{,4}" encuentra cualquier secuencia de hasta cuatro caracteres del mismo rango. |

| | |
|---|---|
| | Hay que tener en cuentra que si se omite el primer número, 0 también es un número válido de repeticiones. Si se especifica un número sólo, busca ese número exacto de repeticiones. |

Además, la función **regex()** cada vez que se ejecuta rellena automáticamente una tabla predefinida global de cadenas llamado ***REGEX_REG[]*** con cada sección entre paréntesis en la expresión original de la cadena localizada, empezando por el índice 1. Esta posibilidad requiere mención aparte, ya que es muy potente, pues permite hacer cosas como buscar una fecha y localizar sus partes. La expresión regular "(..)/(..)/(....)" buscará expresiones del estilo "10/04/1980" y rellena en ***REGEX_REG[1]*** el día, en ***REGEX_REG[2]*** el mes, y en ***REGEX_REG[3]*** el año.

PARÁMETROS:

STRING EXPRESIÓN : Expresión regular
STRING CADENA : Cadena original

VALOR DE RETORNO: INT : Posición encontrada o -1 si no se encontró

Un ejemplo trivial sería éste (el mundo de las expresiones regulares es un universo en sí mismo):

```
import "mod_text";
import "mod_key";
import "mod_regex";

process main()
private
   string a="pepitoholamanolito";
end
begin
/*Busco cadenas de cuatro carácteres de los cuales los dos últimos han de ser "la"*/
   write(0,50,10,4,"Posicion:" + regex("..la",a));
/*Busco cadenas que SÓLO se compongan de los carácteres "ol" (es decir, que antes y
después de ellos haya espacios*/
   write(0,50,20,4,"Posicion:" + regex("^ol$",a));
/*Busco cadenas de cuatro carácteres de los cuales los tres últimos han de ser "ola" y el
primer alguna de las letras a,s o d.*/
   write(0,50,30,4,"Posicion:" + regex("[asd]ola",a));
/*Busco cadenas que sean como "la", "hala", "hahala", "hahahala",etc*/
   write(0,50,40,4,"Posicion:" + regex("(ha)*la",a));
/*Busco cadenas que sean como "hala", "hahala", "hahahala",etc*/
   write(0,50,50,4,"Posicion:" + regex("(ha)+la",a));
/*Busco cadenas "la" o "hala"*/
   write(0,50,60,4,"Posicion:" + regex("(ha)?la",a));
/*Busco cadenas "hahala","hahahala" o "hahahahala"*/
   write(0,50,70,4,"Posicion:" + regex("(ha){2,4}la",a));
/*Busco cadenas "hala" o "hola"*/
   write(0,50,80,4,"Posicion:" + regex("(ha|ho)la",a));
/*Busco cadenas íntegramente formadas por dígitos*/
   write(0,50,90,4,"Posicion:" + regex("[[:digit:]]",a));
repeat
          frame;
until(key(_esc))
end
```

**REGEX_REPLACE(“EXPRESION”,”CAMBIA”,”CADENA”)**

Esta función, definida en el módulo “mod_regex”, busca una expresión regular dentro de una cadena, y sustituye cada coincidencia por el nuevo texto indicado. Es decir, dada una expresión regular, comprueba si la cadena dada la cumple en algún punto, y sustituye lo encontrado por el nuevo texto indicado como tercer parámetro. Devuelve la cadena resultante (no se modifica la original).

Este nuevo texto es una cadena corriente, con la salvedad de que es posible usar la secuencia "\0", "\1", etc, hasta "\9", para sustituir por las secciones entre paréntesis de la expresión regular original, tal cual se encontraron en cada posición. Para ver qué es una expresión regular y qué metacaracteres son admitidos, consulta la documentación de la

función **regex()**

PARÁMETROS:

STRING EXPRESIÓN : Expresión regular
STRING CAMBIA : Cadena con el nuevo texto
STRING CADENA : Cadena original

VALOR DE RETORNO: STRING : Cadena resultante con todos los cambios

Pongamos un ejemplo muy simple simple de uso de **regex_replace()**:

```
import "mod_text";
import "mod_key";
import "mod_regex";

process main()
private
          string result;
end
begin
          result=REGEX_REPLACE("caca","bombon","el caca comia caca con caca y nueces");
          write(0,10,10,0,result);
          repeat
                    frame;
          until(key(_esc))
end
```

Y otro:

```
import "mod_text";
import "mod_key";
import "mod_regex";

process main()
private
          string result;
end
begin
          result=REGEX_REPLACE("ca+","p","el caca comia caca con caca y nueces");
          write(0,10,10,0,result);
          repeat
                    frame;
          until(key(_esc))
end
```

Y otro un poco más complejo:

```
import "mod_text";
import "mod_key";
import "mod_regex";

process main()
private
          string result;
end
begin
          result=REGEX_REPLACE("(ca|co)","p","el caca comia caca con caca y nueces");
          write(0,10,10,0,result);
          repeat
                    frame;
          until(key(_esc))
end
```

Finalmente, comentar la existencia de otras dos funciones de cadena, **split()** y **join()**, cuya finalidad es

respectivamente, partir una cadena en varias cadenas utilizando una determinada expresión regular como separador; y unir varias cadenas en una.

**SPLIT("EXPRESION","CADENA",&ARRAY,TAMAÑO)**

A partir de una expresión regular que actúa como separador, esta función, definida en el módulo "mod_regex", divide la cadena original en partes. Estas partes se almacenarán como elementos del vector de strings indicado como tercer parámetro (precedido del símbolo &). El cuarto parámetro es el número máximo de elementos que podrá almacenar (es decir, la longitud del array en sí).

Por ejemplo, si la cadena original es "12/04/1980" y la expresión regular es "/", la función guarda las cadenas "12", "04" y "1980" en las tres primeras posiciones del array y devuelve 3.

Es importante notar que el separador puede ser una expresión regular compleja. Es posible por ejemplo poner como separador la expresión "[/- ]", la cual dividiría la cadena allí donde hubiera un caracter "/", un caracter "-", o bien un espacio. El uso de expresiones regulares es un concepto avanzado en sí mismo, se recomienda consultar la función **regex()** para obtener más información.

PARAMETROS:
- STRING "Expresión" : Expresión regular
- STRING "Cadena" : Cadena original
- STRING POINTER ARRAY: Vector para guardar los resultados
- INT TAMAÑO : Número de elementos en el vector

VALOR RETORNADO: INT: Número de elementos

Un ejemplo, que parte una cadena en trozos utilizando como separador el símbolo de espacio:

```
import "mod_text";
import "mod_key";
import "mod_regex";

process main()
private
      string cadena="Esto es una cadena de prueba.";
      string mivector[9];
      int elementos,i;
End
begin
elementos=split(" ",cadena,&mivector,9);
for(i=0;i<elementos;i++)
          write(0,100,10+10*i,4,mivector[i]);
end
loop
     frame;
     if(key(_esc)) break;end
end
end
```

**JOIN("SEPARADOR",&ARRAY,TAMAÑO)**

Dado un array de cadenas, esta función, definida en el módulo "mod_regex", junta todas ellas en una cadena final, intercalando la cadena especificada como separador entre los elementos.

Esta función permite obtener el efecto contrario a la función split().Por ejemplo, si un array contiene tres cadenas: "12", "04" y "1980", y se utiliza la cadena "/" como separador, esta función devolverá la cadena "12/04/1980".

PARAMETROS:
- STRING "Separador" : Cadena a intercalar entre elementos
- STRING POINTER ARRAY: Vector de donde obtener las cadenas a unir
- INT TAMAÑO : Número de elementos en el vector

VALOR RETORNADO: STRING: Cadena generada a partir de la unión de los elementos.

Un ejemplo, que crea una cadena a partir de los valores de los elementos de un vector de strings, separándolos entre ellos con el símbolo del espacio:

```
import "mod_text";
import "mod_key";
import "mod_regex";

process main()
private
      string cadena;
      string mivector[9]="Hola","qué","tal","yo","bien","y","tu?";
End
begin
cadena=join(" ",&mivector,9);
write(0,100,100,4,cadena);
loop
     frame;
     if(key(_esc)) break;end
end
end
```

## *Trabajar con ficheros y directorios

Bennu incorpora una familia de funciones -las llamadas "de Entrada/Salida"-que nos permiten gestionar el trabajo con ficheros (crearlos, escribir en ellos, leerlos, etc). Estas funciones nos pueden ser útiles, por ejemplo, a la hora de guardar datos de una partida en un fichero, para poderlos recuperar más adelante, y así poder incluir en nuestro juego la funcionalidad de salvar/recargar partidas.

### FOPEN("FICHERO",MODO)

Esta función, definida en el módulo "mod_file", abre un fichero en disco, operación previa imprescindible para permitir realizar lecturas y manipulaciones posteriores sobre el mismo (empleando otras funciones). Es decir, ésta es una función imprescindible, la primera que se ha de escribir para poder trabajar con un fichero determinado.

Una vez abierto, el puntero de lectura/escritura se posiciona siempre automáticamente al comienzo del fichero.

PARÁMETROS:

STRING FICHERO : Ruta completa del fichero
INT MODO : Modo de apertura. Puede ser uno de los siguientes:

| | | |
|---|---|---|
| **O_READ** | Equivale a 0 | Modo lectura |
| **O_READWRITE**<br>**O_RDWR** | Equivale a 1 | Modo de lectura/escritura |
| **O_WRITE** | Equivale a 2 | Modo de escritura/creación |
| **O_ZREAD** | Equivale a 3 | Similar a O_READ pero para ficheros comprimidos escritos con O_ZWRITE |
| **O_ZWRITE** | Equivale a 4 | Similar a O_WRITE,pero a la vez comprime el fichero utilizando la librería Zlib |

VALOR DE RETORNO: INT : Identificador del fichero abierto

Con el modo O_READ, podremos leer posteriormente el contenido de este archivo desde el principio (con la función correspondiente)

Con el modo es O_READWRITE podremos leer y escribir en ese archivo desde el principio (con la función correspondiente), sobreescribiendo en el segundo caso lo que ya hubiera, pero también, si quisiéramos preservar el

contenido del archivo y escribir nuevos contenidos al final del fichero, añadiéndose a los que ya hubiera, debemos usar O_READWRITE, (aunque deberíamos mover previamente -con la función correspondiente- el puntero al fin de fichero y empezar a escribir desde allí).

Con el modo O_WRITE, todo lo que se escriba en el fichero sobreescribirá completamente el contenido anterior desde el principio porque éste se borra totalmente.

Un fichero abierto en modo O_READ o O_READWRITE debe existir previamente. En caso contrario, la función devolverá 0 (código de error).En cambio, un fichero será creado si no existe previamente al ser abierto en modo O_WRITE.

Las rutas especificadas (en este comando y en general, en todos los comandos de Bennu donde se tengan que escribir rutas como valores de parámetros) se pueden escribir utilizando los separadores de directorios de Windows (\) o de Unix-Linux (/) indistintamente. No obstante, en las rutas absolutas de Windows es necesario indicar una letra de unidad, con lo que se pierde portabilidad. Por tanto se recomienda, en la medida de lo posible, escribir rutas relativas al propio DCB y no rutas absolutas.

**FWRITE(IDFICHERO,VARIABLE)**

Esta función, definida en el módulo "mod_file", guarda datos en formato binario, contenidos previamente en una variable, en un fichero abierto con la función **fopen()** . Esta función sólo puede guardar una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se guardarán secuencialmente (es decir, por orden uno detrás de otro) todos los elementos de la tabla o estructura. No es válido,en cambio, almacenar datos de tipo POINTER.

PARÁMETROS:

INT IDFICHERO : Identificador de fichero devuelto por FOPEN
VAR VARIABLE : Nombre de la variable a guardar

VALOR DE RETORNO: INT : Número de bytes escritos, 0 en caso de error

Vamos a ver un ejemplo de esta última función. Para crear un archivo, previamente inexistente, y escribir en él el contenido de cierta estructura (por ejemplo), haríamos lo siguiente:

```
import "mod_file";

process main()
private
          int i;
          int idfich;
          struct mistruct[2]
                    int a;
                    string b;
          end =1,"hola",2,"que tal",3,"muy bien";
end
begin
          idfich=fopen("pepito.dat",o_write);
          fwrite(idfich,mistruct);
          fclose(idfich);
end
```

Verás que cuando ejecutes este programa aparentemente no habrá pasado nada. Pero fíjate que en la misma carpeta donde está el DCB aparecerá el archivo "pepito.dat" con los datos en binario que almacenaba la estructura. Puedes comprobar el contenido de este archivo abriéndolo por ejemplo con cualquier editor de texto: verás algún simbolo raro correspondiente al campo entero de la estructura, pero podrás observar los valores de cadena perfectamente. El hecho de que se vean estos "símbolos raros" es debido a que **fwrite()** -y **fread()**- trabajan con los datos de forma binaria, no en modo texto.

Poniendo O_READWRITE como valor del segundo parámetro de **fopen()** también habría funcionado, siempre y cuando, eso sí, el archivo exista previamente: con O_READWRITE si el archivo no existe **fopen()** lanza un

error -devuelve 0-. Por otra parte, tanto con O_WRITE como con O_READWRITE, los datos que contuviera anteriormente un fichero abierto (previamente existente) se sobreescriben con los nuevos datos; si éstos no son suficientes para sobreescribir los datos antiguos enteramente, en el fichero pueden quedar restos de los datos antiguos.

En este ejemplo hemos utilizado una tabla de estructuras, pero en la práctica lo más habitual es utilizar estructuras simples.

Por si alguna vez lo ves en algún código, es bueno saber que la función **fwrite()**, al igual que **fread(),** tiene otro prototipo (se puede observar utilizando la utilidad *moddesc*), que no es usada prácticamente nunca, consistente en tres parámetros de entrada por este orden: un puntero a la variable a guardar (en **fread()**, a leer), el tamaño de dicha variable y el identificador del fichero a guardar (en **fread()**, a leer).

**FREAD(IDFICHERO,VARIABLE)**

Esta función, definida en el módulo "mod_file", lee datos binarios de un fichero abierto con la función **fopen()** y los asigna a una variable para poder así trabajar en el código fuente con la información obtenida.

Si por ejemplo la variable es de tipo INT, este tipo de datos ocupa 4 bytes, por lo que se leerán 4 bytes del archivo. Si es de tipo WORD se leerán 2 bytes, si es de tipo BYTE se leerá uno, y así.

Esta función sólo puede leer una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se leerán cada vez todos los elementos de la tabla o estructura secuencialmente (es decir, por orden uno detrás de otro).

Una característica importante de su funcionamiento es que cada vez que se le invoque, seguirá leyendo a partir de la última posición donde el puntero se quedó en la última lectura.Es decir, que si se realiza una lectura y se rellena una tabla con dicha operación, la siguiente vez que se llame a **fread()** se seguirá leyendo del archivo y se volverá a rellenar la tabla con nuevos valores (los siguientes).

PARÁMETROS:

INT IDFICHERO : Identificador de fichero devuelto por FOPEN
VAR VARIABLE : Nombre de la variable a leer

VALOR DE RETORNO: INT : Número de bytes leídos, 0 en caso de error

Como curiosidad, si se deseara procesar ficheros binarios genéricos o escritos con otra utilidad, puede usarse esta función para leer un dato de tipo BYTE cada vez, y procesarlos en secuencia.

**FEOF(IDFICHERO)**

Esta función, definida en el módulo "mod_file", comprueba si quedan datos por leer de un fichero.

Devuelve 1 si el puntero de lectura/escritura traspasa el final de un fichero abierto con la función **fopen()**, o 0 en caso contrario.

Puede usarse en un bucle que emplee **fread()** para leer datos de un fichero, hasta que ya no queden más en el mismo.

PARÁMETROS: INT IDFICHERO : Identificador de fichero devuelto por FOPEN

VALOR DE RETORNO: INT : 1 si se llegó al final del fichero

De **fread()** y de **feof()** veremos ejemplos posteriormente.

**FSEEK(IDFICHERO,POSICION, DESDE)**

Esta función, definida en el módulo "mod_file", cambia la posición del puntero de lectura/escritura en un fichero abierto con la función **fopen().** La nueva posición viene dada por la combinación de los valores del segundo y tercer parámetro.

El tercer parámetro puede indicar tres puntos concretos, que puede ser el comienzo del fichero (valor 0 ó **SEEK_SET**), la posición actual del puntero –cuyo valor se puede obtener con **ftell()** si se desea- (valor 1 ó **SEEK_CUR**) o el final del fichero (valor 2 ó **SEEK_END**).

El segundo parámetro es el número de bytes que el puntero se moverá, a partir de la posición base indicada en el tercer parámetro, para realmente llegar al sitio deseado. Su valor será positivo si el movimiento del puntero es hacia adelante dentro del contenido del fichero o negativo si es para atrás. Evidentemente, si partimos de la posición base al comienzo del fichero, el segundo parámetro no podrá ser negativo, y si partimos del final de fichero, no podrá ser positivo.

En todo caso el valor retornado por esta función será la nueva posición real en bytes, siempre respecto al origen del fichero. La función **fseek()** no funciona si el archivo está comprimido y por tanto se ha tenido que abrir con O_ZREAD.

PARÁMETROS:

INT IDFICHERO : Identificador de fichero devuelto por **fopen()**
INT POSICIÓN : Nueva posición deseada
INT DESDE : Tipo de posición utilizada. Tal como se ha dicho, puede ser:

| | | |
|---|---|---|
| **SEEK_SET** | Equivale a 0 | A partir del comienzo del fichero |
| **SEEK_CUR** | Equivale a 1 | A partir de la posición actual |
| **SEEK_END** | Equivale a 2 | A partir del final del fichero |

VALOR DE RETORNO: INT : Nueva posición de lectura/escritura, en número de bytes desde el comienzo del fichero (empezando en 0)

Existe una función, definida también en el módulo "mod_file", llamada **frewind(),** la cual tiene un único parámetro que es un identificador de fichero devuelto por **fopen()**, y cuya ejecución es equivalente a *fseek(idfichero,0,SEEK_SET);*. Es decir, cambia la posición del puntero de lectura/escritura al principio del fichero siempre.

**FTELL(IDFICHERO)**

Esta función, definida en el módulo "mod_file", devuelve la posición en número de bytes del puntero de lectura/escritura de un fichero abierto con la función **fopen()** , siempre respecto al origen del fichero (0 para indicar el comienzo).

Se suele utilizar junto con **fseek()**, para situar el puntero en un lugar concreto a partir del lugar donde está actualmente.

PARÁMETROS: INT IDFICHERO : Identificador de fichero devuelto por FOPEN

VALOR DE RETORNO: INT : Posición de lectura/escritura, en número de bytes desde el comienzo del fichero (empezando en 0)

Un ejemplo de **fseek()** y **ftell()** (necesitarás un fichero llamado "pepito.dat" -lo creamos en el ejemplo de **fwrite() -**, aunque no importa su contenido -incluso puede estar vacío-):

```
import "mod_text";
import "mod_file";
import "mod_key";
import "mod_say";
import "mod_proc";

process main()
private
        int idfich;
```

```
end
begin
        idfich=fopen("pepito.dat",o_read);
        if(idfich == 0) say("Fichero no encontrado"); exit(); end
/*Muestro  en qué posición -byte- estoy.Como acabo de abrir el archivo, debo de estar al principio,o sea, en el byte 0*/
        write(0,10,10,4,ftell(idfich));
//Desde el principio del fichero, me muevo un byte para adelante
        fseek(idfich,1,0);
/*Vuelvo a mostrar donde estoy. Como me he movido un byte desde el principio, ha de mostrar un 1*/
        write(0,10,20,4,ftell(idfich));
//Desde donde estoy -a un byte del principio-, me vuelvo a mover un byte para adelante
        fseek(idfich,1,1);
/*Vuelvo a mostrar donde estoy. Como desde donde estaba -que era en el byte 1- me he movido otro byte, ftell mostrará
un 2,  desde el principio.*/
        write(0,10,30,4,ftell(idfich));
        fclose(idfich);
        repeat
                frame;
        until(key(_esc))
end
```

| **FLENGTH (IDFICHERO)** |
|---|
| Esta función, definida en el módulo "mod_file", devuelve el tamaño total, en bytes, de un fichero abierto con la función **fopen()**<br><br>Esta función no es válida para archivos abiertos con O_ZREAD o O_ZWRITE<br><br>PARÁMETROS: INT IDFICHERO : Identificador de fichero devuelto por FOPEN<br><br>VALOR DE RETORNO: INT : Número de bytes que ocupa el fichero. |

Un ejemplo de esta función (necesitarás un fichero llamado "pepito.dat"):

```
import "mod_text";
import "mod_file";
import "mod_key";
import "mod_say";
import "mod_proc";

process main()
private
    int fp,long;
end
begin
    fp = fopen("pepito.dat", O_READ);
    if (fp==0) say("Error en intentar abrir el fichero"); exit(); end
    long = flength(fp);
    fclose(fp);
    write(0,120,80,4,"La length del fichero es "+ long);
    while(!key(_ESC)) frame; end
end
```

| **FCLOSE(IDFICHERO)** |
|---|
| Esta función, definida en el módulo "mod_file", cierra un fichero abierto previamente por **fopen()**. Todos los ficheros abiertos deben cerrarse una vez acabadas las operaciones de lectura o escritura deseadas, ya que existe un límite al número de ficheros abiertos simultáneamente, en función del sistema operativo.<br><br>PARÁMETROS : INT IDFICHERO : Identificador de fichero devuelto por **fopen()** |

Un ejemplo útil sería el de añadir contenido nuevo a este archivo existente sin sobreescribir el contenido

que ya había. Si te fijas en el código de ejemplo de **fwrite()** que pusimos unos párrafos más arriba, verás que siempre, todas las veces que se ejecutaba el programa, se empezaba a escribir desde el principio del fichero. Para modificar este deberemos abrir el archivo (el mismo del ejemplo de **fwrite()**: "pepito.dat",el cual ya tiene contenido escrito) obligatoriamente en modo O_READWRITE y lo primero, situarnos justo al final para poder empezar a escribir desde allí. O sea:

```
import "mod_file";

process main()
private
          int i;
          int idfich;
          struct mistruct[2]
                    int a;
                    string b;
          end = 1,"ah, si?",2,"pues yo",3,"también";
end
begin
          idfich=fopen("pepito.dat",o_write);
          fseek(idfich,0,2);
          fwrite(idfich,mistruct);
          fclose(idfich);
end
```

Fíjate que lo único que hemos cambiado respecto el primer ejemplo de **fwrite()** es el modo de apertura del archivo (insisto: con O_WRITE no podemos añadir contenido), y utilizar antes de realizar ninguna acción de escritura, la orden **fseek()**. En este caso, **fseek()** colocará el puntero en el final (3r parámetro) del archivo "pepito.dat" (1r parámetro), y a partir de ahí se moverá 0 bytes (2º parámetro) –o sea, que se quedará en el final-.

Ahora por fin podríamos hacer un programa que leyera el contenido de "pepito.dat" y lo mostrara por pantalla. Pero nos surge un problema. ¿Cuántas estructuras contiene "pepito.dat"? Depende de cuántas veces se hayan escrito datos en él. En el archivo puede haberse escrito un número indeterminado de estructuras. Si ejecutamos por ejemplo dos veces el código anterior, tendremos en "pepito.dat" seis estructuras simples. Pero nada impide volver a ejecutar ese código otra vez, con lo que tendríamos 9 estructuras para leer. ¿Cómo sabremos que tenemos que leer 9 estructuras, ni más ni menos, de ese archivo? Un manera sería así:

```
import "mod_text";
import "mod_file";
import "mod_key";

process main()
private
  int i=10;
  int idfich;
  struct mistruct
          int a;
          string b;
  end
end
begin
  idfich=fopen("pepito.dat",o_read);
  //Leo las estructuras que contiene el fichero una a una hasta llegar al final del mismo
  loop
     fread(idfich,mistruct);
     if (feof(idfich)!=false) break; end
     write(0,50,i,4,mistruct.a);
     write(0,100,i,4,mistruct.b);
     i=i+10;
     frame;
  end
  fclose(idfich);

  repeat
          frame;
  until(key(_esc))
end
```

Fíjate que aquí ya usamos estructuras simples, que es lo habitual. Lo único que hacemos es ir repitiendo la lectura del archivo estructura a estructura –y poniendo sus valores por pantalla- hasta que ya no queden más por leer. En cada lectura se sobreescribe el contenido de la variable *mistruct*, así que ésta contendrá en cada momento los valores de la estructura acabada de leer del fichero. ¿Y cómo sabemos que ya no quedan más estructuras por leer? Cuando llegamos al final del fichero, y esto ocurre cuando la función **feof()** devuelve 1 (true).

Aunque a lo mejor se te había ocurrido en vez de realizar el bucle LOOP/END, hacer lo mismo pero con un bucle WHILE/END así:

```
while(feof(idfich)==false)
          fread(idfich,mistruct);
          write(0,50,i,4,mistruct.a);
          write(0,100,i,4,mistruct.b);
          i=i+10;
          frame;
end
```

Pero este código tiene un pequeño defecto: muestra los datos bien pero también muestra un último carácter extraño que sobra. ¿Por qué muestra "de más"? Por que fíjate lo que hemos cambiado. La condición del "traspaso" del fin de fichero se comprueba ANTES de leer la siguiente estructura, de tal manera que cuando el programa lee la última estructura, en la siguiente comprobación **feof()** continúa valiendo falso porque aunque se haya llegado al final, no se ha "traspasado" todavía:se está en el umbral. Por eso, en esa comprobación, al continuar **feof()** valiendo falso, se ejecuta otra vez el interior del while. Y como en ese momento ya no hay nada que leer porque ya sí que se ha "traspasado" por completo el fin de fichero, **fread()** fracasa totalmente y devuelve resultados totalmente erráticos. Por eso, te doy un consejo: comprueba si se ha "traspasado" el fin de fichero justo DESPUÉS de leer con **fread()**: de esta manera, si has te has pasado del final, lo sabrás de inmediato y el programa podrá reaccionar a tiempo.

Hasta ahora, los ejemplos vistos sólo son capaces de leer o escribir una única estructura en un archivo. Pero si queremos almacenar más de una estructura, ¿tendremos que crear un archivo diferente para cada una de ellas? No. El siguiente ejemplo muestra cómo, con un poquito de maña, es posible almacenar en un único archivo el número de estructuras diferentes que se necesite (en el ejemplo, dos), utilizando para ello gran parte de las funciones explicadas hasta ahora, como **fwrite()**, **fread()**, **ftell()** o **fseek()**. La idea del programa es básicamente la siguiente: a partir de tener un par de estructuras rellenas de valores, se guardan éstos en un archivo, seguidamente se borran de las estructuras en memoria (y se ve que eso es así) y finalmente se recuperan del archivo donde se guardaron al principio. Los comentarios del código son muy completos, léelos:

```
import "mod_text";
import "mod_file";
import "mod_key";
import "mod_proc";

type struct1
          string strng;
          int intgr;
end
type struct2
          int var1;
          float var2;
end


process main()
private
          int fp;
          int i;
          struct1 st1[6] ="s10",10,"s5",5,"s210",210,"s3",3,"s33",33,"s12",12,"s3",3;
         struct2 st2[6] = 10,10.10,5,5.5,210,210.210,3,3.3,33,33.33,12,12.12,3,3.3;
   /*Estructura que nos servirá para almacenar, para cada estructura a guardar en el
archivo, a qué distancia en bytes se encuentra del comienzo de éste. La idea es que cada
estructura se almacenará completa una seguida de la otra, de tal manera que luego al leer
el archivo, se necesitará saber en qué posición dentro de él comienza la estructura que
se desea recuperar. Así pues, header.oStruct1 nos indicará la posición de comienzo de la
primera estructura y header.oStruct2 lo mismo con la segunda. Si deseáramos escribir más
```

```
estructuras dentro del archivo, deberíamos de incluir un campo más por cada una de ellas
dentro de esta estructura header*/
          struct header
                    int oStruct1 = 0;
                    int oStruct2 = 0;
           end
end

begin
//Abro el fichero para escribir en él el contenido de dos estructuras
fp = fopen("test007.dat", O_WRITE);
if (!fp) //Si hay algún error...
           write(0,10,180,0,"ERROR al crear archivo test007.dat");
           write(0,30,190,0,"presione ESC para salir");
           while(!key(_esc)) frame; end //Hasta que no se pulse ESC no pasa nada
           while(key(_esc)) frame; end //Mientras se está pulsando, tampoco
           exit();
end
/*La primera estructura que se va a escribir al comienzo del fichero siempre será la
estructura header, y luego, se empezarán a escribir las demás. Pero los valores de los
campos de la estructura header son la distancia en bytes de las demás estructuras
respecto el origen del fichero, y esta información no se sabe hasta que se hayan escrito
dichas estructuras en el fichero. Por lo que la línea siguiente lo que hace es saltar "el
cabezal" dentro del fichero un número de bytes igual al tamaño de header para empezar a
escribir más adelante, y así dejar reservado ese espacio del principio para que
finalmente se pueda llenar de contenido*/
fseek(fp, sizeof(header), SEEK_SET);
/*La primera estructura se escribirá justo después de que acaben los datos de la
estructura header, por lo que al haber saltado con la línea anterior hasta este punto
precisamente, ya sabemos en qué posición comenzarán los datos de str1: simplemente
tenemos que usar ftell para que nos devuelva la posición del cabezal, y este valor lo
almacenaremos en el campo correspondiente dentro de header*/
header.oStruct1 = ftell(fp);
//Escribimos de un tirón el contenido de str1
fwrite(fp, st1);
/*Una vez acabados de escribir todos los datos de str1, hemos llegado al punto del
archivo donde se tienen que empezar a escribir los datos de str2. Pero antes, almacenamos
en header la posición inicial de str2*/
header.oStruct2 = ftell(fp);
//Escribimos de un tirón el contenido de str2
fwrite(fp, st2);
//Situamos el cabezal al principio del fichero otra vez
fseek(fp, 0, SEEK_SET);
/*Ahora que ya sabemos las posiciones iniciales de str1 y str2 (header.oStruct1 y
header.oStruct2), podemos rellenar de valor el espacio inicial del fichero que dejamos
vacío al principio. Y ¿para qué queremos almacenar aquí las posiciones iniciales de las
diferentes estructuras en el fichero? Para que en la lectura posterior del archivo, a
partir de esta información del principio, se pueda distinguir cada dato a qué estructura
pertenece.*/
fwrite(fp, header);
fclose(fp);

//Muestro por pantalla los datos almacenados en un único archivo de dos estructuras
write(0,0,0,0,"Datos almacenados");
/*Para saber el número de registros que existen en un determinado momento del TDU st1 (y
así realizar las iteraciones pertinentes) el truco está en dividir el tamaño total en
bytes del TDU entero -sizeof(st1)- entre el tamaño en bytes de un registro, típicamente
el primero -sizeof(st1[0])-. Esto dará el número de registros que tiene esa estructura en
ese momento*/
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
           write(0,0,10+i*10,0,st1[i].strng+" , "+st1[i].intgr);
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
           write(0,160,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
end

//Borro los datos de las dos estructuras en memoria
write(0,10,190,0,"presione ESC para limpiar los datos");
```

```
while(!key(_esc)) frame; end
while(key(_esc)) frame; end
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
          st1[i].strng = "";
          st1[i].intgr = 0;
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
          st2[i].var1 = 0;
          st2[i].var2 = 0.0;
end

//Muestro por pantalla que efectivamente estos datos han sido borrados (de la memoria)
delete_text(0);
write(0,0,0,0,"Datos borrados");
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
          write(0,0,10+i*10,0,st1[i].strng+" , "+st1[i].intgr);
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
          write(0,160,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
end

/*Recupero esos datos perdidos de las dos estructuras a partir de la copia guardada en
fichero al principio del programa*/
write(0,10,190,0,"presione ESC para recuperar los datos");
while(!key(_esc)) frame; end
while(key(_esc)) frame; end
fp = fopen("test007.dat", O_READ);
if (!fp)
          write(0,10,180,0,"ERROR al abrir archivo test007.dat");
          write(0,30,190,0,"presione ESC para salir");
          while(!key(_esc)) frame; end
          while(key(_esc)) frame; end
          exit();
end
/*Lo primero que hago, una vez abierto el fichero, es mirar en qué posiciones dentro de
él comienzan cada una de las dos estructuras guardadas, para poderlas leer sin problemas.
Str1 comenzará donde marque header.oStruct1 y Str2 comenzará donde marque
header.oStruct2*/
fread(fp, header);
//Me sitúo al comienzo de la primera estructura y la leo toda completa
fseek(fp, header.oStruct1, SEEK_SET);
fread(fp, st1);
//Me sitúo al comienzo de la segunda estructura y la leo toda completa
fseek(fp, header.oStruct2, SEEK_SET);
fread(fp, st2);
fclose(fp);

/*Muestro otra vez por pantalla los datos contenidos en las  dos estructuras, recuperados
a partir del fichero. Se puede comprobar que son los mismos valores que los mostrados al
principio del programa.*/
delete_text(0);
write(0,0,0,0,"Datos recuperados");
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
          write(0,0,10+i*10,0,st1[i].strng+" , "+st1[i].intgr);
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
          write(0,160,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
end

write(0,0,190,0,"Presione ESC para salir");
while(!key(_ESC)) frame; end
while(key(_ESC)) frame; end
delete_text(0);
end
```

Con **fwrite()** y **fread()** logramos escribir y leer ficheros en formato binario (es decir, que pueden contener datos numéricos, tablas ó estructuras), pero si únicamente deseamos trabajar con ficheros de texto, Bennu dispone de funciones especialmente pensadas para escribir y leer este tipo de ficheros, como las siguientes:

**FPUTS(IDFICHERO,"TEXTO")**

Esta función, definida en el módulo "mod_file", escribe una cadena de texto en un fichero abierto en modo de escritura. Sirve para crear ficheros de texto, al contrario que **fwrite()**, que puede usarse para crear ficheros binarios.

Cada cadena se guarda en el fichero en una línea aparte. Si la cadena contiene un salto de línea o más, se guardará el carácter \ al principio de cada uno (lo cual permite volver a leerla empleando en una sola llamada).

PARÁMETROS:

INT IDFICHERO : Identificador de un fichero abierto con **fopen()**
STRING TEXTO : Texto a escribir

**FGETS(IDFICHERO)**

Esta función, definida en el módulo "mod_file", lee la siguiente línea de un fichero de texto abierto en modo de lectura, devolviéndola en una cadena, sin incluir el carácter o caracteres de final de línea.

Es decir, cada vez que se llame a esta función, leerá (únicamente) la siguiente línea de texto del fichero, por lo que escrita dentro de un bucle podría utilizarse para leer un archivo desde el el principio hasta el final. La manera de detectar que se ha llegado al final del archivo es la misma que antes: con **feof()**

Una línea que acaba con el carácter \ en el fichero original será juntada con la siguiente, sin incluir dicho caracter, pero incluyendo un carácter de salto de línea en su posición. Esto permite almacenar y recuperar cadenas arbitrarias en disco, incluso aunque contengan saltos de línea.

PARÁMETROS: INT IDFICHERO : Identificador de un fichero abierto con FOPEN

VALOR DE RETORNO: STRING : Texto de la siguiente línea

Miremos un par de ejemplos de estas dos últimas funciones. Si quisiéramos crear un archivo de texto llamado "pepito.txt" , podríamos hacer algo muy parecido a esto:

```
import "mod_file";

process main()
private
          int idfich;
end
begin
          idfich=fopen("pepito.txt",o_write);
          fputs(idfich,"hola, ¿qué tal?. Muy bien.");
          fputs(idfich,"¿Ah,sí? Pues yo también");
          fclose(idfich);
end
```

De igual manera que en ejemplos anteriores, si quisiéramos añadir nuevo texto a un archivo que ya tuviera, tendríamos que cambiar el modo de apertura a O_READWRITE y utilizar la orden **fseek()**

Para leer un archivo de texto, lo que tendríamos que hacer algo parecido a lo siguiente:

```
import "mod_file";
import "mod_key";
import "mod_text";

process main()
private
          int idfich;
          string cadenatotal="";
          string cadenaleida;
end
```

```
begin
          idfich=fopen("pepito.txt",o_read);
          loop
                    if(feof(idfich)!=false) break; end
                    cadenaleida=fgets(idfich);
                    cadenatotal=cadenatotal+cadenaleida;
          end
          fclose(idfich);
          write(0,160,50,4,cadenatotal);
          repeat
                    frame;
          until(key(_esc))
end
```

Fíjate cómo vamos concatenando en cada iteración la cadena recientemente leída a la cadena ya creada, actualizando de esta manera la propia cadena "total". Éste es el método típico para ir añadiendo porciones de texto a una cadena que por lo tanto va aumentando de longitud.

Otro ejemplo de **fputs()** y **fgets()** similar al anterior es el siguiente:

```
import "mod_file";
import "mod_key";
import "mod_text";

process main()
private
    string linea[] = "ésta es la primer línea...",
                     "ésta es la segunda...",
                     "ésta parece ser la tercera...",
                     "creo que ya vamos por la cuarta...",
                     "ésta es la última";
    int i,fp;
end
begin
  fp = fopen("pepito.txt", O_WRITE);
/*Recordemos que si sizeof(linea) devuelve el tamaño total de la tabla y sizeof(linea[0])
devuelve el tamaño de un elemento de esa tabla -en este caso el primero, pero puede ser
cualquiera ya que todos son iguales-, la división sizeof(linea)/sizeof(linea[0]) nos
devolverá el número de elementos llenos que tiene la tabla en el momento actual.Así pues,
este bucle recorre cada elemento de la tabla "linea", que resulta ser una cadena, la cual
irá siendo guardada en un archivo de texto*/
    for(i=0;i<sizeof(linea)/sizeof(linea[0]);i++)
        fputs(fp,linea[i]);
    end
    fclose(fp);
    write(0,0,0,0,"Recuperando usando fgets");
    fp = fopen("pepito.txt", O_READ);
    i=0;
    while(!feof(fp))
    //Recupero en cada iteración una nueva línea
        write(0,0,20+i*10,0,fgets(fp));
        i++;
    end
    fclose(fp);
    while(!key(_esc)) frame; end
end
```

El siguiente ejemplo utiliza una función definida por nosotros para sobreescribir una línea concreta de un fichero de texto dado por una frase dada:

```
import "mod_file";
import "mod_say";
import "mod_text";

process main()
begin
          file_putline("pepito.txt",5,"hola hola hola");
```

```
end

//LineNumber empieza por 1, claro
function file_putline(string tfile, int linenumber,string text)
private
          int i;
          int idfile;
          int poslineaantes;
          int poslineadespues;
          string cadenavacia="";
end
begin
idfile=fopen(tfile,O_READWRITE);
if(idfile==0)
          say("Fichero no encontrado");
          return;
end

//Sitúo el puntero al principio de la linea pedida
while (i < LineNumber - 1)
          if(feof(idfile)!=0)
                    say("El fichero tiene menos líneas de las que has indicado");
                    return;
          end
          fgets(idfile);
          i++;
end

/*Calculo la longitud actual de la cadena que se sobreescribirá, y creo una cadena vacía
de esa misma longitud. Esto lo hago porque si la cadena antigua fuera más larga que la
nueva, al sobreescribirla aparecerían los restos sobrantes de la antigua al final sin
sobreescribir. "Tapándola" primero con una línea en blanco, este efecto ya no aparecerá*/
poslineaantes=ftell(idfile);
fgets(idfile);
poslineadespues=ftell(idfile);
for(i=0;i<(poslineadespues-poslineaantes);i++)
          cadenavacia=cadenavacia + " ";
end

//Me vuelvo a situar al principio de la línea y sobreescribo la línea, primero con la
cadena vacía y después con la cadena que quiero guardar*/
fseek(idfile,poslineaantes,0);
fputs(idfile,cadenavacia);
fseek(idfile,poslineaantes,0);
fputs(idfile, text);
fclose(idfile);
end
```

Existe un problema no resuelto con el código anterior y es que **fputs()** siempre añade un salto de línea a la cadena que escribe, por lo que al sobreescribir la línea escogida, añadiremos un salto de línea al fichero que no debería aparecer. ¿Eres capaz de resolver este comportamiento no deseado?

Otras funciones interesantes referentes al manejo de ficheros son las siguientes:

| **SAVE("FICHERO",VARIABLE)** |
|---|
| Esta función, definida en el módulo "mod_file",guarda datos en formato binario, contenidos previamente en una variable, en un fichero, cuya ruta/nombre viene dado por su primer parámetro. Es decir, realiza la misma tarea que **fwrite()**. No obstante, aporta una diferencia: esta función no necesita tener abierto previamente el fichero -con **fopen()**- : es ella quien lo abre transparentemente para escribir en él sin necesidad de usar ninguna otra función.<br><br>Al igual que con **fwrite()**, esta función sólo puede guardar una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se guardarán secuencialmente (es decir, por orden uno detrás de otro) todos los elementos de la tabla o estructura.No es válido,en cambio, almacenar datos de tipo POINTER. |

PARÁMETROS:

STRING FICHERO : Ruta del fichero a escribir
VAR VARIABLE : Nombre de la variable a guardar

Un ejemplo de su posible uso:

```
import "mod_file";
import "mod_key";
import "mod_text";

process main()
private
  struct dat[9]
    int a,b,c;
  end
end
begin
  write(0,160,100,4,"1) Salvar los valores de la estructura en archivo");
  repeat
    if(key(_1))
      from x=0 to 9;
        dat[x].a=x+1;
        dat[x].b=x+1;
        dat[x].c=x+1;
      end
      save("save.dat",dat);
    end
    frame;
  until(key(_esc))
end
```

**LOAD("FICHERO",VARIABLE)**

Esta función lee datos binarios de un fichero cuya ruta/nombre viene dado por su primer parámetro, y los asigna a una variable para poder así trabajar en el código fuente con la información obtenida. Si por ejemplo la variable es de tipo INT (es decir, de 4 bytes), se leerán 4 bytes del archivo, si es de tipo WORD se leerán 2 bytes, si es de tipo BYTE se leerá uno, etc. Es decir, realiza la misma tarea que **fread()**. No obstante, aporta una diferencia: esta función no necesita tener abierto previamente el fichero: es ella quien lo abre transparentemente para leer de él.

Al igual que con **fread()**, esta función sólo puede leer una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se leerán cada vez todos los elementos de la tabla o estructura secuencialmente (es decir, por orden uno detrás de otro).

PARÁMETROS:

STRING FICHERO : Ruta del fichero a escribir
VAR VARIABLE : Nombre de la variable a leer

Un ejemplo de su posible uso:

```
import "mod_file";
import "mod_key";
import "mod_text";

process main()
private
  struct dat[9]
    int a,b,c;
  end
end
begin
  //Valores iniciales de la estructura (al no estar inicializada, serán todo 0)
  from x=0 to 9;
```

```
    write_var(0,60,x*10+85,4,dat[x].a);
    write_var(0,160,x*10+85,4,dat[x].b);
    write_var(0,260,x*10+85,4,dat[x].c);
  end
  write(0,1,40,3,"1)Leer fichero creado antes en el ejemplo de SAVE");
  write(0,10,65,3,"Valores del fichero:");
  repeat
    if(key(_1))
      load("save.dat",dat);
    end
    frame;
  until(key(_esc))
end
```

| **FILE_EXISTS("FICHERO")** |
|---|
| Esta función, definida en el módulo "mod_file", comprueba si un fichero existe o no en la ruta especificada.<br><br>Devuelve 1 si el fichero existe y 0 si no existe. El fichero se busca en la ruta actual si no se escribe ningún valor al parámetro o en la ruta que se haya especificado.<br><br>Un "alias" para esta función es **fexists()**<br><br>PARÁMETROS: STRING FILE : Ruta completa del fichero |

| **FILE ("FICHERO")** |
|---|
| Esta función, definida en el módulo "mod_file", abre el fichero de la ruta especificada, devuelve su contenido en una única cadena y cierra el fichero. Por lo tanto, para su correcto funcionamiento es aconsejable que el fichero sea de texto.<br><br>PARAMETROS: STRING FILE: Ruta completa del fichero (ha de existir previamente)<br><br>VALOR DEVUELTO: STRING : Cadena cuyo valor será todo el contenido del fichero. |

Un ejemplo de estas dos últimas funciones:

```
import "mod_file";
import "mod_say";

process main()
private
  int i;
  string s;
end
begin
  if(file_exists("pepito.txt")) s=file("pepito.txt"); end
  say(s);
end
```

No obstante, si en vez de utilizar **say()** utilizas **write()** para mostrar el texto, puedes encontrarte con que si el contenido del fichero incluye saltos de línea y retornos de carro -es decir, si has pulsado ENTER mientras lo escribías-, lo que se visualizará por pantalla, en vez de éstos, serán unos caracteres extraños (en realidad, los caracteres correspondientes al salto de línea y retorno de carro en la tabla ASCII -cuyos código numéricos son el 13 y el 10, respectivamente- ). Si quieres que por pantalla se vean los saltos de línea/retorno de carro igual que como están en el archivo, hay que trabajar un poco más. El siguiente código soluciona este problema, pero hace uso de funciones no vistas todavía de tratamiento de cadenas (**find(), chr(), substr()**), que serán vistas -y explicadas en profundidad- en el apartado correspondiente, más adelante en este mismo capítulo:

```
import "mod_file";
import "mod_key";
import "mod_string";
import "mod_text";
```

```
process main()
begin
    readFileWithCRLF("pepito.txt");
    while(!key(_esc)) frame; end
end

function readFileWithCRLF(string ruta)
private
    string  sVar = "" ;
    int i,fin,fila,aux,long;
end
begin
    //Obtengo todo el texto del fichero
    sVar = file(ruta);
    fila = 0;
    //Guardo la longitud total del texto obtenido
    long = len(sVar);
    //Voy carácter a carácter
    for (i=0;i<long;i++)
        //Busco el caracter de salto de línea
        fin = find(sVar,chr(13),i);
        //Si no lo encuentro...
        if (fin!=-1)
            write(0,0,fila*10,0,substr(sVar,i,fin-i));
        //...y si sí
        else
            write(0,0,fila*10,0,substr(sVar,i));
            break;
        end

        while(fin<long)
            aux=asc(substr(sVar,fin,1));
            if (aux==10 || aux==13)
                    fin++;
                    if (aux==13)  fila++;  end
              else
                    break;
              end
        end
        i=fin;
    end
end
```

En estos momentos es posible que te sea difícil comprender el interior de la función readFileWithCRLF() ("CR" quiere decir "Carriage Return" -es decir, retorno de carro, carácter ASCII 10- y "LF" quiere decir "Line Feed" -es decir, salto de línea, carácter ASCII 13-). Lo más interesante, de todas maneras, es poder aprovechar el código tal cual de la función en cualquier otro proyecto que tengas entre manos, y aprovechar así la funcionalidad que aporta.

| **RM("FICHERO")** |
|---|
| Esta función, definida en "mod_dir", borra el fichero indicado por el parámetro "fichero". Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar el borrado.<br><br>También se puede especificar la ruta completa en el parámetro y el fichero será borrado utilizando la ruta completa en lugar de borrarse del directorio actual. Las rutas pueden indicarse usando el separador de directorios / o \ indiferentemente.<br><br>En plataformas que así lo permitan, un fichero no se podrá borrar si no se tienen los permisos adecuados para ello.<br><br>PARÁMETROS: STRING FICHERO : Nombre (o ruta) del fichero a borrar |

Como ejemplo de la función anterior, realizaremos un código que cree un archivo (mediante **save()** mismo, compruebe que existe mediante **fexists()**, lo borre con **rm()**, y vuelva a comprobar que ya no existe, otra vez con **fexists()**.

```
import "mod_file";
import "mod_text";
import "mod_proc";
import "mod_key";
import "mod_dir";

process main()
private
  int i=1;
end
begin
  save("prueba.dat",i);
  if(fexists("prueba.dat"))
          while(!key(_space))
                    delete_text(0);
                    write(0,0,0,0,"El fichero existe");
                    frame;
          end
  else
          while(!key(_space))
                    delete_text(0);
                    write(0,0,0,0,"ERROR en grabar");
                    frame;
          end
  end
  rm("prueba.dat");
  if(!fexists("prueba.dat"))
          while(!key(_enter))
                    delete_text(0);
                    write(0,0,0,0,"El fichero NO existe");
                    frame;
          end
  else
          while(!key(_enter))
                    delete_text(0);
                    write(0,0,0,0,"ERROR en crear");
                    frame;
          end
  end
end
```

Una función que hace exactamente lo mismo que **rm()** es **fremove()**. Tiene el mismo único parámetro (el nombre o ruta del fichero a eliminar); podrías modificar el código anterior sustituyendo **rm()** por **fremove()** y comprobarás que no ocurre ninguna diferentecia. No obstante, **fremove()** está definida en el módulo "mod_file".

Otra función interesante es **fmove()**:

| **FMOVE("FICHERO1","FICHERO2")** |
|---|
| Esta función, definida en el módulo "mod_file", mueve el fichero cuyo nombre o ruta (absoluta o relativa a donde esté situado el DCB) se especifica en el primer parámetro, a la carpeta cuya ruta (absoluta o relativa a donde esté situado el DCB) se especifica en el segundo parámetro. Es muy importante incluir en este segundo parámetro, al final de la ruta de la carpeta de destino, el nombre del fichero también: si se escribe tan sólo la ruta de la carpeta de destino, **fmove()** devolverá error. Al escribir, por tanto, el nombre del fichero en el segundo parámetro, podemos aprovechar esta circunstancia para renombrarlo también. De hecho, se puede utilizar **fmove()** para renombrar ficheros sin necesidad de moverlos de carpeta, de forma análoga a como se hace frecuentemente en muchos sistemas operativos con los correspondientes comandos de consola.<br><br>Si la acción ha ocurrido correctamente, esta función devolverá 0. Si ha ocurrido algún error (el archivo de origen no existe, la carpeta destino no existe, como segundo parámetro se ha escrito la ruta de una carpeta sin nombre de fichero final, etc) devolverá -1.<br><br>PARÁMETROS:<br>STRING FICHERO1 : Nombre o ruta del fichero que se desea mover |

| STRING FICHERO2 : Nueva ruta del fichero (incluyendo el nombre de éste, nuevo también o no) |
|---|

Un ejemplo rápido:

```
import "mod_file";
import "mod_say";

process main()
private
        int err;
end
begin
        err=fmove("pepito.txt","ruta/carpeta/pepito.txt");
        say(err);
end
```

Otras funciones, que tienen que ver más con el manejo de directorios,son:

| **CD()** |
|---|
| Esta función, definida en el módulo "mod_dir", devuelve el nombre del directorio actual (el directorio donde reside el DCB). Este será el directorio donde actuarán por defecto todas las operaciones con ficheros como pueden ser **fopen()** o **save()** (es decir, en el caso que no se especifiquen las rutas completas de los archivos como parámetro de dichas funciones y sólo se especifique el nombre).<br><br>VALOR DE RETORNO: STRING: Nombre del directorio actual |

| **CHDIR("DIRECTORIO")** |
|---|
| Esta función, definida en el módulo "mod_dir", cambia el directorio actual por defecto al que se especifica con el parámetro "directorio". Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar el cambio al nuevo directorio, por ejemplo, si el directorio indicado no existe.<br><br>PARÁMETROS: STRING DIRECTORIO : Nuevo directorio |

| **MKDIR("DIRECTORIO")** |
|---|
| Esta función, definida en el módulo "mod_dir", crea un nuevo directorio, indicado por el parámetro "directorio", en el directorio actual. Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar la creación del nuevo directorio, por ejemplo si el directorio ya existiese.<br><br>También se puede especificar la ruta completa en el parámetro y el directorio será creado utilizando la ruta completa en lugar de crearse a partir del directorio actual. Las rutas pueden indicarse usando el separador de directorios / o \ indiferentemente.<br><br>Un directorio no podrá crearse si existe un fichero, recurso o directorio con el mismo nombre o si la ruta especificada es incorrecta. En los sistemas que así lo permitan, no se podrán crear directorios en la ruta indicada si no se tienen los permisos adecuados.<br><br>PARÁMETROS: STRING DIRECTORIO : Nuevo directorio |

| **RMDIR("DIRECTORIO")** |
|---|
| Esta función, definida en el módulo "mod_dir", borra el directorio indicado por el parámetro "directorio" en el directorio actual. Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar la creación del nuevo directorio, por ejemplo si el directorio ya existiese.<br><br>También se puede especificar la ruta completa en el parámetro y el directorio será creado utilizando la ruta completa en lugar de crearse a partir del directorio actual. Las rutas pueden indicarse usando el separador de directorios / o \ indiferentemente. |

Un directorio no podra ser borrado si es el directorio actual, si no está vacio o si es el directorio raíz del sistema. En plataformas que así lo permitan, un directorio no se podrá borrar si no se tienen los permisos adecuados para ello.

PARÁMETROS: STRING DIRECTORIO : Directorio a borrar

Un ejemplo de estas últimas funciones podría ser:

```
import "mod_dir";
import "mod_key";
import "mod_text";

process main()
private
   string a;
end
begin
   mkdir("./midir");
   chdir("./midir");
   a=cd();
   write(0,0,0,0,a);
   chdir("../");
   rmdir("./midir");
   repeat
           frame;
   until(key(_esc))
end
```

Acuérdate de que tanto en Windows como en Linux, la carpeta ".." significa la carpeta inmediatamente superior a la actual, y la carpeta "." significa la carpeta actual.

**GLOB("PATRON")**

Esta función, definida en el módulo "mod_dir", busca cualquier fichero en el directorio actual cuyo nombre coincida con el patrón indicado.

El patrón es un nombre de fichero en el cual el comodín "?" indica "cualquier carácter" y el comodín "*" indica "uno o más caracteres cualesquiera".Un uso habitual es "*.*" para buscar todos los ficheros presentes. También es posible indicar un directorio relativo (por ejemplo,"FPG/*.*") además del patrón.

Esta función devuelve el nombre del primer fichero que encuentra y, en sucesivas llamadas con el mismo patrón, va devolviendo el resto de ficheros que coincidan con el patrón. Cuando ya no queden más ficheros, o tras la primera llamada si no se encontró ninguno, devuelve una cadena en blanco. El nombre de fichero que va devolviendo esta función no contiene el directorio.

Esta función además rellena, cada vez que es llamada, la estructura global predefinida ***FILEINFO***, que podemos utilizar para obtener la siguiente información:

| | |
|---|---|
| **FILEINFO.PATH** | Directorio donde se encuentra el fichero |
| **FILEINFO.NAME** | Nombre completo del fichero |
| **FILEINFO.DIRECTORY** | Vale 1 (TRUE) si el fichero es un directorio, o 0 (FALSE) si es un fichero corriente |
| **FILEINFO.HIDDEN** | Vale TRUE si el fichero está oculto |
| **FILEINFO.READONLY** | Vale TRUE si el fichero no tiene activados permisos de escritura |
| **FILEINFO.SIZE** | Tamaño en bytes del fichero |
| **FILEINFO.CREATED** | Cadena de texto que contiene la fecha y hora de creación del fichero |
| **FILEINFO.MODIFIED** | Cadena de texto que contiene la fecha y hora del último acceso al fichero |

PARÁMETROS: STRING PATRÓN : Patrón de búsqueda

VALOR DE RETORNO: STRING : Nombre de siguiente fichero o "" si no quedan más.

Como ejemplo de esta última función tienes el siguiente código:

```
import "mod_dir";
import "mod_key";
import "mod_text";

process main()
Private
          String archivo; //cadena donde se guarda el nombre del archivo
          Int cont=0;
end
Begin
          //seleccionamos una carpeta, que es la superior a la actual
          chdir("../");
          //buscamos un archivo cuyo nombre respete el patrón indicado
          archivo=glob("*.???");
          While (archivo!="") //comprobamos que ha encontrado algun archivo
              Write(0,10,cont*10,0, archivo); //escribimos el nombre del archivo…
              Write(0,250,cont*10,0,FILEINFO.CREATED);//y su fecha y hora de creación
              cont=cont+1; //avanzamos una linea
              archivo=glob("*.???"); //buscamos otro archivo
          End
          //y esperamos que se pulse escape para salir
          Loop
                    If (Key(_ESC)) break; end
                    Frame;
          End
End
```

Una funcionalidad similar, aunque no idéntica, a **glob()** la ofrece el trío de funciones siguiente:

**DIROPEN("PATRON")**

Esta función, definida en el módulo "mod_dir", busca cualquier fichero ó directorio cuyo nombre ó ruta (absoluta ó relativa a la carpeta donde está ubicado el DCB) coincida con el patrón indicado, para poder ser leído/s posteriormente por **dirread()**. Devolverá un identificador de lista de ficheros -y directorios- encontrados (que será usada por **dirread()** -y **dirclose()** -), o bien 0 si se ha producido algún error (por ejemplo, al no existir el directorio).

El patrón es un nombre de fichero ó directorio en el cual el comodín "?" indica "cualquier carácter" y el comodín "*" indica "uno o más caracteres cualesquiera". De esta manera, **diropen()** podrá abrir más de un archivo ó directorio a la vez, con lo que el identificador que devuelve en realidad no es el de un fichero concreto, sino el de la lista que contiene los ficheros que concuerdan con el patrón especificado como parámetro. La función **dirread()** se encargará, a cada llamada que se le haga, de ir recorriendo uno a uno los elementos presentes en esa lista dada por el identificador devuelto por **diropen()**.

Un valor habitual del parámetro de esta función es, por ejemplo, "*" para buscar todos las carpetas y ficheros presentes (a partir de la carpeta donde está situado el DCB). También es posible indicar un rutas absolutas.

PARÁMETROS: STRING DIRECTORIO : Nombre ó ruta del directorio a abrir

VALOR DE RETORNO: INT : Identificador del directorio, ó 0 si error.

**DIRREAD(IDDIR)**

Esta función, definida en el módulo "mod_dir", devuelve el nombre del primer archivo ó directorio que encuentra dentro de la lista devuelta por **diropen()** a través de IDDIR y, en sucesivas llamadas con el mismo identificador, va devolviendo el resto de archivos ó directorios que queden en la lista. Cuando ya no queden más elementos, o tras la primera llamada si no se encontró ninguno, devuelve una cadena en blanco. El nombre de archivo ó directorio que va devolviendo esta función no contiene su ruta absoluta.

Esta función además rellena, cada vez que es llamada, la misma estructura global predefinida ***FILEINFO*** comentada cuando se vio la función **glob()**.

PARÁMETROS: INT IDDIR : Identificador de la lista de elementos a leer,, obtenido de **diropen()**

VALOR DE RETORNO: STRING : Nombre de siguiente fichero/directorio ó "" si no quedan más.

**DIRCLOSE(IDDIR)**

Esta función, definida en el módulo "mod_dir", cierra una lista de directorios (especificado por su identificador) previamente abierto con **diropen()**

PARÁMETROS: INT IDDIR: Identificador del directorio a cerrar, obtenido en su momento de **diropen()**

Como ejemplo de estas tres últimas funciones , aquí va el siguiente código, que realiza prácticamente lo mismo que el ejemplo de **glob()**. En este caso se listan los archivos de la carpeta actual (donde está ubicado el DCB) que tengan extensión "html".

```
import "mod_dir";
import "mod_key";
import "mod_text";

process main()
Private
          String fichero; //cadena donde se guarda el nombre del fichero
          int cont=0;
          int iddir;
end
Begin
        //buscamos los archivos y directorios cuyo nombre respete el patrón indicado
          iddir=diropen("*.html");
          fichero=dirread(iddir);
          While (fichero!="") //comprobamos que ha encontrado algun archivo o directorio
              Write(0,10,cont*10,0, fichero); //escribimos el nombre del archivo…
              Write(0,250,cont*10,0,FILEINFO.CREATED);//y su fecha y hora de creación
              cont=cont+1; //avanzamos una linea
              fichero=dirread(iddir); //buscamos otro archivo
          End
          dirclose(iddir);

          //y esperamos que se pulse escape para salir
          Loop
                    If (Key(_ESC)) break; end
                    Frame;
          End
End
```

La ventaja que tiene utilizar las tres funciones anteriores respecto **glob(),** es que se separa el hecho de crear la lista de elementos que concuerdan con el patrón del hecho de irlos recorriendo uno a uno. Por ello, las funciones dirXXX se pueden usar de forma recursiva: es decir, con ellas se pueden recorrer a la vez varios directorios, ya que se utilizan diferentes identificadores independientes para cada directorio.

## *Trabajar con temporizadores

Un temporizador es un cronómetro programable accesible en todo momento. Bennu dispone de 10 temporizadores accesibles a través de una tabla de variables globales enteras llamadas TIMER (desde timer[0] a timer[9]), siempre y cuando se importe el módulo "mod_timers". Estos temporizadores siempre se inicializan automáticamente al comenzar el programa al valor 0 y se incrementan automáticamente 100 veces por segundo. En cualquier momento puede cambiarse el valor de uno de ellos, por ejemplo inicializándolo de nuevo a 0, para controlar

una cantidad de tiempo concreta.

Ya sabes que las variables globales se actualizan cada instrucción frame. Por tanto, es obvio que los valores leídos a cada frame en un temporizador no serán consecutivos a menos que nuestro programa se esté ejecutando a 100 frames por segundo.Es decir: supongamos que el programa de ejecuta a 40 fps. Eso quiere decir que en un segundo pasarán 40 frames, por lo que el valor del temporizador cambiará 40 veces. No obstante, en un segundo el temporizador siempre aumenta en 100 su valor, por lo que en un segundo el temporizador habrá aumentado 100 unidades pero haciéndolo en 40 saltos.

Una cosa importante que comentar también es que para realizar las comprobaciones de los valores que tienen los temporizadores en un momento dado (por ejemplo, con un *if(timer[x]==limite_tiempo)* ) debemos utilizar siempre con los operadores de >= o > y nunca con el == ya que no se puede asegurar que tras la actualización del temporizador el valor sea exactamente igual al deseado.

Vamos a poner un ejemplo. Queremos que un nivel de un juego dure 5 segundos, y se quiere mostrar por pantalla el tiempo que queda. Habrá una variable global llamada *tiempo* que almacenará esos segundos, y un proceso "controlTiempo()", que la modifica. En este ejemplo, cuando se llegue al final de la cuenta atrás, el programa termina su ejecución, pero se podrían inventar otras reacciones. Aquí está el código:

```
import "mod_text";
import "mod_key";
import "mod_proc";
import "mod_timers";

process main()
begin
          controlTiempo();
          loop
                    if(key(_esc)) exit(); end
                    frame;
          end
end

process controlTiempo()
private
          int tiempo;
end
begin
          timer[0] = 0;
          write_var(0,100,100,4,tiempo);
          loop
                    tiempo = 5 - (timer[0] / 100);
                    if (tiempo == 0) exit(); end
                    frame;
          end
end
```

Fíjate lo que hace el proceso "controlTiempo()". Cada vez que se ejecuta, resetea el valor de la variable *timer[0]* (el único temporizador que utilizaremos). Entonces se entra en un bucle que lo que hace es decrementar el valor de *tiempo*, hasta que valga 0, momento cuando se acaba la partida. Lo interesante está en la línea *tiempo=100-(timer[0] / 100);*. Esta línea lo que debería de hacer es decrementar en una unidad el valor de *tiempo* cada segundo. Para controlar que este decremento se haga efectivamente cada segundo recurrimos al timer[0]. Cada segundo hemos dicho que timer[0] (y cualquier timer de los diez que hay, aunque no se utilicen) se incrementa 100 unidades. Por lo tanto, en el primer segundo *tiempo* valdrá 100–(100/100)=100-1=99; en el segundo segundo *tiempo* valdrá 100– (200/100)=100-2=98, etc.

A continuación escribo un pequeño y sencillo programa que registra cuánto tarda el jugador en apretar una tecla. Este código se podría ampliar de forma que se convirtiera por ejemplo en un juego donde el jugador tuviera que apretar dicha tecla dentro de unos márgenes de tiempo establecidos (no pudiendo demorarse o adelantarse demasiado), o incluso guardar los tiempos marcados en un fichero en disco para lograr hacer una lista de récords con todos los jugadores…son ideas para desarrollar.

```
import "mod_text";
```

```
import "mod_key";
import "mod_proc";
import "mod_timers";

process main()
Begin
  record();
  Loop
       If(key(_esc)) exit();End
       Frame;
  End
End

Process record()
private
  int mitiempo=0;
  int txttimer;
end
Begin
  write(0,0,0,0,"PULSA ENTER para PARAR EL TIMER");
  write(0,0,10,0,"TU TIEMPO: ");
  txttimer=write_var(0,15,20,1,timer[0]);
  Repeat
          mitiempo=timer[0];
        Frame;
  Until(key(_enter))
  delete_text(txttimer);
  write(0,80,10,0,mitiempo);
End
```

Otro ejemplo muy curioso, donde se utilizan los temporizadores en combinación con funciones gráficas como **xput()** y **map_put_pixel()** es el siguiente. En él se demuestra que con un poco de imaginación, se pueden lograr efectos realmente espectaculares:

```
import "mod_video";
import "mod_map";
import "mod_rand";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_draw";
import "mod_timers";

PROCESS Main()
PRIVATE
          int i,c,map,tim;
END
BEGIN
    set_mode(320,200,32);
    map=new_map(320,200,32);map_clear(0,map,rgb(0,0,0));
    repeat
                c=rgb(rand(0,255),rand(0,255),rand(0,255));
//El temporizador se divide entre 5 para que el zoom-in/zoom-out no sea tan rápido
                tim = (timer[0]/5) %100;
                IF (tim > 50) tim = 100-tim; END
                xput (0,map,160,100,timer[0]*-50, 200+tim, 4, 0) ;
                FROM i = 0 TO 100;
                    map_put_pixel(0,map,rand(0,320),rand(0,200),c) ;
                    map_put_pixel(0,map,rand(0,320),rand(0,200),1) ;
                END
                FRAME;
    until(key(_esc))
    unload_map(0,map) ;
END
```

Hay que saber, por último, que los timers funcionan de forma independiente de todo lo que demás. Asi que si, por lo que sea, el juego se ralentiza, en el ejemplo anterior la partida durará los 100 segundos, pero para el juego

habría pasado menos tiempo: supongamos que, en condiciones normales el juego ejecuta 1000 frames en los 100 segundos, en caso de ralentizarse ejecutaria 950 frames en ese tiempo: el protagonista avanza menos, los enemigos avanzan menos, hay menos disparos...Esto es lo que se llama un problema de sincronización.

Además de la variable ***TIMER***, en Bennu podemos utilizar un nuevo tipo de temporizador, con precisión de milésimas de segundos. Su funcionamiento es muy similar al de los timers clásicos, pero con la diferencia (aparte de la precisión), de que sólo existe un timer de este tipo (en vez de los 10 anteriores) y se accede a él no a partir de elementos de un vector predefinido sino a partir de la función **get_timer()**, definida en el módulo "mod_time" (no confundir con "mod_timers"). Esta función, la cual no tiene parámetros, sirve para obtener de ella el valor que el temporizador tenga en un instante en particular. Este temporizador se pone en marcha automáticamente al iniciarse el programa y no se parará hasta que éste sea finalizado. Veamos este ejemplo, donde se puede ver el contador en plena acción con precisión de milésimas:

```
import "mod_text";
import "mod_key";
import "mod_proc";
import "mod_time";

process main()
private
          int tiempo;
end
begin
          loop
                    if(key(_esc)) exit(); end
                    tiempo=get_timer();
                    write_var(0,100,100,4,tiempo);
                    frame;
          end
end
```

Si quisiéramos mostrar sólo con precisión de segundos, no tendríamos más que dividir *tiempo* entre 1000.

A partir de aquí, podemos elegir según nuestras necesidades si continuamos haciendo servir los clásicos timers de precisión de centésimas o esta nueva función **get_timer()**.

No obstante, hay que tener en cuenta que la función **get_timer()** NO sirve para asignar un valor concreto al temporizador: eso no lo podemos hacer, a diferencia de los timers clásicos, donde la línea *timer[0]=0;* resetea a cero ese temporizador. Con **get_timer()** es incorrecto por ejemplo escribir *get_timer()=0*, ya que no es posible reestablecer a mano un valor de este temporizador. Entonces, ¿cómo lo hacemos para tener el control sobre un período de tiempo dado? Pues el truco está en recoger al principio de ese período en una variable el valor que en ese instante tenga el temporizador, y luego, al final de ese período, volver a recoger el nuevo valor y restarle el del principio. De esta manera, la diferencia que obtenemos es el número de milisegundos que han transcurrido entre un instante y el otro. Por ejemplo, aquí mostramos el primer ejemplo que vimos en este apartado, pero esta vez utilizando **get_timer()** en vez de timer[0]:

```
import "mod_text";
import "mod_key";
import "mod_proc";
import "mod_time";

process main()
begin
          controlTiempo();
          loop
                    if(key(_esc)) exit(); end
                    frame;
          end
end

process controlTiempo()
private
          int comienzo;
        int tiempo;
```

```
end
begin
          comienzo=get_timer(); //Instante inicial
          write_var(0,100,100,4,tiempo);
          loop
                    tiempo = 5 - (get_timer()- comienzo)/1000;
                    if (tiempo == 0) exit(); end
                    frame;
          end
end
```

Otro ejemplo, para acabar de entender el sistema de temporizadores:

```
import "mod_text";
import "mod_key";
import "mod_proc";
import "mod_time";
import "mod_map";

process main()
begin
  write(0,10,10,0,"Pulsa (1) para esperar 1 segundo");
  write(0,10,20,0,"Pulsa (2) para esperar 2 segundos");
  loop
     if (key(_esc)) exit(); end
     if (key(_1)) ExampleImage(1000); end
     if (key(_2)) ExampleImage(2000); end
     frame;
  end
end

process ExampleImage(int duracion)
private
   int comienzo;
   int tiempo;
end
begin
   graph=new_map(30,30,32); map_clear(0,graph,rgb(255,255,0));
   x=160;y=120;
   comienzo= get_timer(); //Guardo el tiempo inicial
/*Comienza a contar el tiempo, desde el momento actual.Mientras el tiempo transcurrido
sea, a partir del comienzo, menor que la duración establecida, no hago nada. Cuando ya
haya pasado ese tiempo, saldré del for, y al acabar el proceso, su gráfico que se estaba
mostrando desaparecerá. Atención al valor del paso de la siguiente iteración: se vuelve a
recoger el nuevo tiempo actual.Se podría haber usado un while en vez de un for. Y también
podría haber escrito dentro de este for todo aquello que deseara que se ejecutara sólo en
el periodo de tiempo dado por "duración"*/
   for(tiempo=get_timer();tiempo<comienzo+duración;tiempo=get_timer())
        delete_text(0);
        write(0,10,50,0,tiempo-1000);
        frame;
   end
end
```

Fíjate que en cada ejecución del código anterior, la variable *tiempo* finaliza en un valor aproximado cerca del valor final que debe tener (es decir, si hemos marcado la opción "1", no acabará en 1000, sino unas veces unas milésimas por encima y otras veces unas milésimas por debajo). Tenlo en cuenta.

¿Para qué sirve, en el código siguiente, el proceso "retraso()"?

```
import "mod_text";
import "mod_key";
import "mod_proc";
import "mod_time";
import "mod_map";

process main()
```

```
private
          int idproc;
end
begin
  write(0,0,10,0,"Pulsa 1 para dormir, 2 para despertar, 3 para matar");
  idproc=proceso1();
  loop
     if (key(_esc)) exit(); end;
     if (key(_1)) retraso(2000,idproc,s_sleep); end
     if (key(_2)) retraso(2000,idproc,s_wakeup); end
     if (key(_3)) retraso(2000,idproc,s_kill); end
     frame;
  end
end

process proceso1()
begin
graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,255));
x=160; y=120;
loop
          frame;
end
end

process retraso(int duracion,int idproc,int senal)
private
 int time_init;
 int time_end;
end
begin
time_init = get_timer();
time_end = time_init + duracion;
repeat frame; until (get_timer() >= time_end)
signal(idproc,senal);
end
```

Una manera diferente de implementar una función “retraso()” similar al ejemplo anterior, pero sin utilizar ningún tipo de temporizadores (sólo con el uso de la variable predefinida *frame_time*) y más genérica, ideal para cualquier tipo de situaciones, podría ser la siguiente:

```
import "mod_text";
import "mod_key";

Function int retraso(float seconds)
Begin
    While( (seconds=seconds-frame_time) > 0 ) frame; End
End

process main()
Begin
    Repeat
          if(key(_space))
                  retraso(3); //Se espera 3s antes de continuar
                     write(0,0,0,0,"Texto");
          end
          if(key(_enter)) delete_text(0); end
        frame;
    Until(key(_ESC))
End
```

Y otra pregunta más: ¿qué diferencia hay entre los dos códigos siguientes?

```
import "mod_text";
import "mod_key";
import "mod_time";

process main()
```

```
private
           int min=2; //minutos
           int seg=4; //segundos
           int nFps=10; //Numero de frames a los que cambiará el valor de seg
           int contador; //contador
End
begin
write_var(0,100,100,4,min);
write(0,108,100,4,":");
write_var(0,120,100,4,seg);
loop
           if(key(_esc))break;end
           if(contador==nFps)
                      seg--;
                      contador=0;
                      if(seg<0)
                                 min--;
                                 seg=59;
                      end
           end
        if(min==0 And seg==0) break;end
        contador++;
        frame;
end
end
```

Y:

```
import "mod_text";
import "mod_key";
import "mod_time";

process main()
private
           int min=2; //minutos
           int seg=4; //segundos
          int tiempo;
End
begin
write_var(0,100,100,4,min);
write(0,108,100,4,":");
write_var(0,120,100,4,seg);
tiempo=get_timer();
loop
           if(key(_esc))break;end
           if(get_timer()>1000+tiempo)
                      seg--;
                      if(seg<0)
                                 min--;
                                 seg=59;
                      end
                      tiempo=get_timer();
           end
        if(min==0 and seg==0) break;end
        frame;
end
end
```

Efectivamente, la velocidad de cambio en los "segundos" en el primer código depende del FPS que lleve ese programa (en concreto, el cambio se produce cada 10 frames), y en el segundo código sí que va sincronizado realmente con los segundos reales ya que utiliza un temporizador. Uno podría pensar que si se establece una velocidad de 10 frames/segundo (con *set_fps(10,0);*), los dos códigos podrían ser equivalentes, pero esto no es exactamente así, porque Bennu trata de ejecutar 10 frames/segundo -en este caso- PERO NO lo garantiza: no tiene suficiente precisión para asegurar que esta velocidad sea así ni constante. Esto depende de la máquina, depende la cantidad de procesos activos,etc,etc. Por eso, el uso de temporizadores siempre es más fiables ya que son independientes de todo esto. Aunque usar timers es un arma de doble filo: se tiene precisión en tiempo real en cualquier máquina, pero si el

ordenador es lento, los personajes se moverán más despacio y será más dificil que lleguen a una meta en el tiempo previsto (es como si el tiempo fuera más rápido). Por eso es recomendable usar los timers si se necesita un reloj o el juego no depende del tiempo, pero es mejor usar variables contadoras si lo que se quiere hacer es un juego contrarreloj.

## *Trabajar con fechas

Bennu no tiene demasiada flexibilidad a la hora de permitirnos utilizar funciones predefinidas que trabajen con fechas.De hecho, sólo incorpora dos.

**TIME()**

Esta función, definida en el módulo "mod_time", devuelve la fecha actual en formato entero como el número de segundos transcurridos desde el 1 de Enero de 1970.

El valor devuelto por esta función debería utilizarse con la función **ftime()** para transformarla a un formato legible.

**FTIME("FORMATO",TIME)**

Esta función, definida en el módulo "mod_time", devuelve una cadena de texto con el formato de fecha especificado en el primer parámetro, a partir de la fecha en formato entero que recibe como segundo parámetro. La fecha en formato entero corresponde al número de segundos transcurridos desde el día 1 de Enero de 1970, por ejemplo generada por la función **time()**.

Para especificar el formato se le pasa a la función una cadena donde todos los carácteres de texto se respetarán y las siguientes secuencias de carácteres se transforman acorde al siguiente listado:

**%d**: Día, dos dígitos (01-31)
**%m**: Mes, dos dígitos (01-12)
**%y**: Año, dos dígitos (00-99)
**%Y**: Año, cuatro digitos
**%H**: Hora, dos dígitos (00-23)
**%M**: Minuto, dos dígitos (00-59)
**%S**: Segundo, dos dígitos (00-59)
**%e**: Día, uno o dos dígitos (1-31)
**%k**: Hora, uno o dos dígitos (0-23)
**%a**: Nombre abreviado del día
**%A**: Nombre completo del día
**%b**: Nombre abreviado del mes
**%B**: Nombre completo del mes
**%C**: Centuria, dos dígitos (19-99)
**%I**: Hora americana, dos dígitos (01-12)
**%l**: Hora americana, uno o dos dígitos (1-12)
**%p**: Coletilla americana de la hora (AM/PM)
**%P**: Coletilla americana de la hora (am/pm)
**%u**: Día de la semana (1-7)
**%U**: Día de la semana empezando por 0 (0-6)
**%j**: Día del año, tres dígitos (001-366)
**%U**: Número de semana del año, dos dígitos (00-53)

PARÁMETROS:

STRING FORMATO : Formato en el que se presentará la fecha
INT TIME : Fecha en formato entero, obtenido normalmente a partir de la función time()

Un ejemplo sencillo de uso de ambas funciones sería:

```
import "mod_text";
```

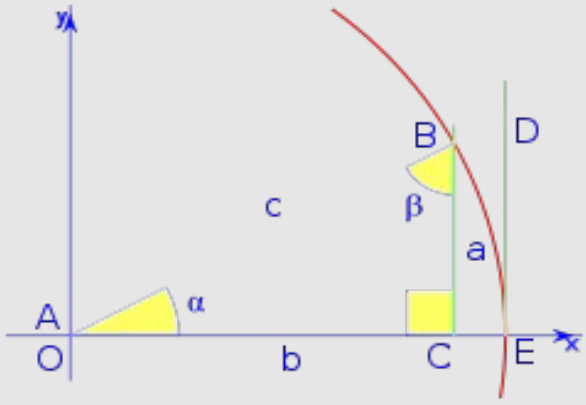
```
import "mod_key";
import "mod_time";

process main()
begin
  write(0,160,390,4,"Pulsa ESC para Salir...");
  write(0,10,60,3,"%d - Día, dos dígitos:   "+ftime("%d",time()));
  write(0,10,70,3,"%e - Día, uno o dos dígitos:   "+ftime("%e",time()));
  write(0,10,80,3,"%m - Mes, dos dígitos:   "+ftime("%m",time()));
  write(0,10,90,3,"%y - Año, dos dígitos:   "+ftime("%y",time()));
  write(0,10,100,3,"%Y - Año, cuatro dígitos:   "+ftime("%Y",time()));
  write(0,10,110,3,"%H - Hora, dos dígitos:   "+ftime("%H",time()));
  write(0,10,120,3,"%k - Hora, unos o dos dígitos:   "+ftime("%k",time()));
  write(0,10,130,3,"%M - Minuto, dos dígitos:   "+ftime("%M",time()));
  write(0,10,140,3,"%S - Segundo, dos dígitos:   "+ftime("%S",time()));
  write(0,10,150,3,"%j - Día del año, tres dígitos:   "+ftime("%j",time()));
  write(0,10,160,3,"%U - Numero de semana del año:   "+ftime("%U",time()));
  repeat
    frame;
  until(key(_esc))
end
```

## *Trabajar con las matemáticas

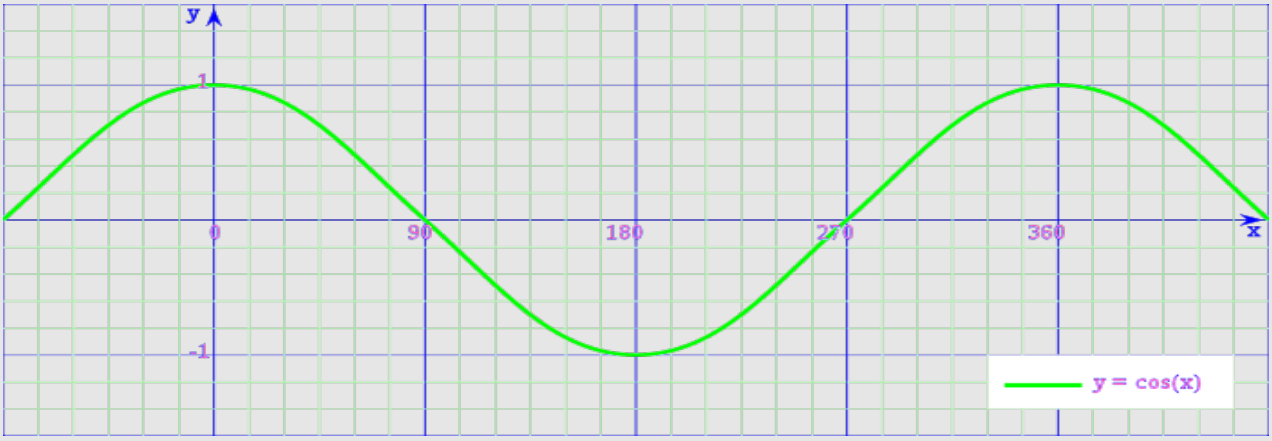
Bennu aporta un conjunto (limitado) de funciones matemáticas para poder realizar cálculos aritméticos básicos. Algunas son:

| **ABS(NUMERO)** |
|---|
| Esta función, definida en el módulo “mod_math”, devuelve el valor absoluto de un número dado, es decir, convierte cualquier número negativo en positivo y deja números positivos o el 0 inalterados.<br><br>PARÁMETROS: FLOAT NÚMERO : Número en coma flotante |

| **COS(NUMERO)** |
|---|
| Esta función, definida en el módulo “mod_math”, devuelve el coseno de un ángulo dado en milésimas de grado (la unidad estándar para especificar ángulos en Bennu). El resultado será un número en coma flotante entre -1 y 1.<br><br>Recuerda que el coseno de un ángulo se puede definir como la razón entre el cateto adyacente (b) y la hipotenusa (c) del triángulo rectángulo generado por dicho ángulo. Es decir: $cos(\alpha)=b/c$ |




Su representación gráfica en el plano xy, donde los valores en el eje x vienen dados en grados sexadecimales, es así:

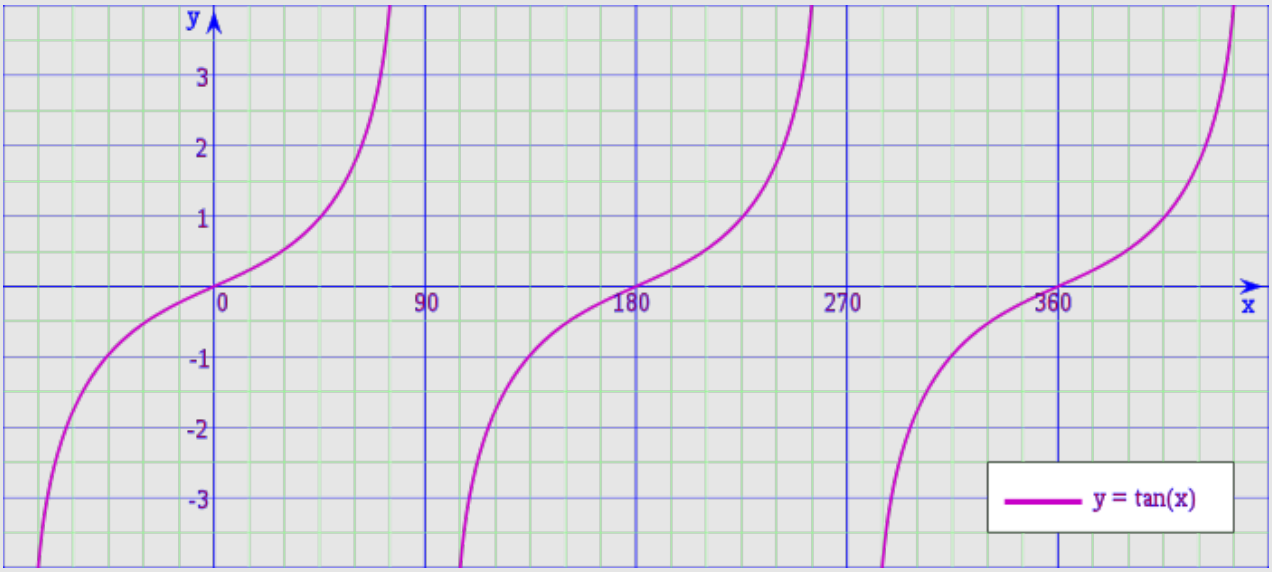



PARÁMETROS: FLOAT NÚMERO : Ángulo en milésimas de grado (90000 = 90º)

De igual forma exiten en Bennu las funciones **SIN(NUMERO)**, que calcula el seno de un ángulo (recordemos que el seno de un ángulo se puede definir como la razón entre el cateto opuesto (a) y la hipotenusa (c) del triángulo rectángulo generado por dicho ángulo. Es decir: *sen(α)=a/c*), y **TAN(NUMERO)**, que calcula su tangente (recordemos que la tangente de ún ángulo se puede definir como la razón entre el cateto opuesto (a) y el cateto adyacente (b) del triángulo rectángulo generado por dicho ángulo. Es decir, *tan(α)=a/b* ) Ambas funciones también tienen como parámetro un número FLOAT , igualmente devuelven un número FLOAT y están definidas en el módulo "mod_math".

La representación gráfica en el plano xy de la función seno, donde los valores en el eje x vienen dados en grados sexadecimales, es así:

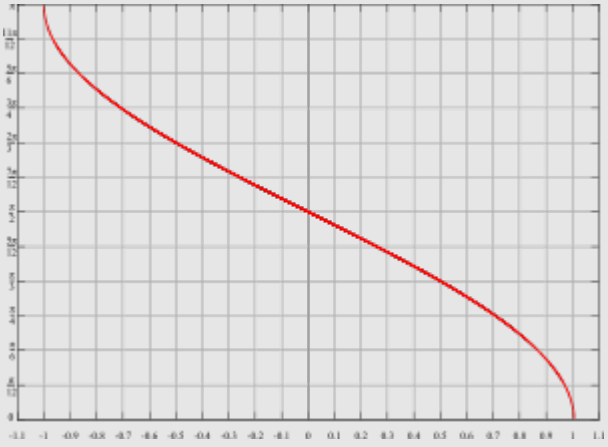
Y la de la función tangente es:




**ACOS(NUMERO)**

Esta función,, definida en el módulo "mod_math", devuelve el arcocoseno de un valor dado como parámetro. (Este valor ha de ser un número decimal entre -1 y 1). Es decir, devuelve un ángulo en milésimas de grado, entre 0 y 180000 cuyo coseno equivale al valor indicado como parámetro. Este resultado devuelto será un número en coma flotante.

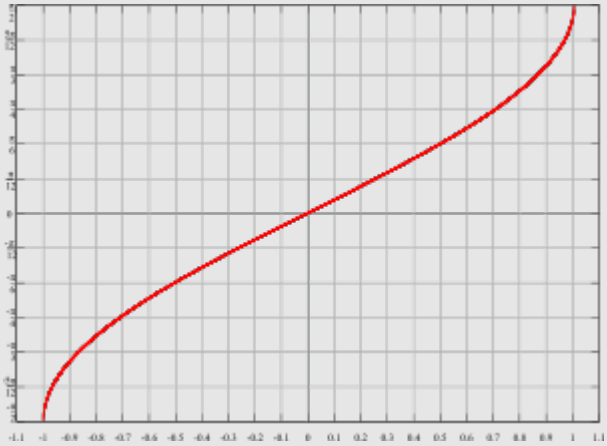
Su representación gráfica sería así:



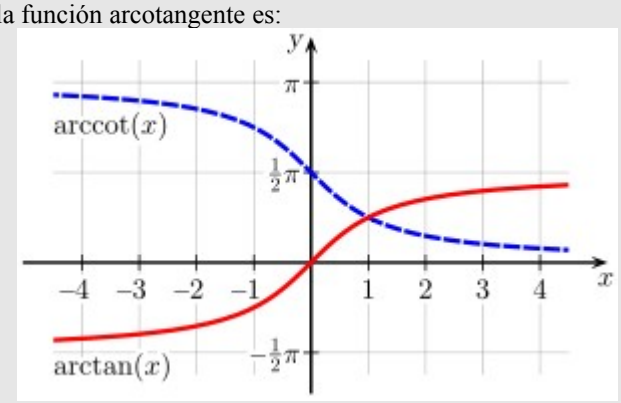
PARÁMETROS: FLOAT NÚMERO : Valor entre -1.0 y 1.0

De igual forma exiten en Bennu las funciones **ASIN(NUMERO)**, que calcula el arcoseno (función inversa del seno) de un número entre -1.0 y 1.0, y **ATAN(NUMERO)**, que calcula su arcotangente (función inversa de la tangente). Ambas también están definidas en el módulo "mod_math".

La representación gráfica de la función arcoseno es:



La representación gráfica de la función arcotangente es:




| **SQRT(NUMERO)** |
|---|
| Esta función, definida en el módulo "mod_math", devuelve la raíz cuadrada, en coma flotante, de un número especificado como parámetro.<br><br>PARÁMETROS: FLOAT NÚMERO : Número en coma flotante |

| **POW(NUMERO, POTENCIA)** |
|---|
| Esta función, definida en el módulo "mod_math", devuelve el resultado, en coma flotante, de elevar un número a una potencia dada. Por ejemplo, el resultado de **pow(2,3)** es 8 (2*2*2). Escribir **pow(49,0.5)** sería lo mismo que **sqrt(49)**.<br><br>PARÁMETROS:<br><br>FLOAT NÚMERO : Número en coma flotante<br>FLOAT POTENCIA : Potencia a la que se desea elevar |

El funcionamiento de las funciones anteriores es trivial. No creo necesario ningún ejemplo de su uso porque resulta evidente.

| **Función de redondeo** |
|---|
| Bennu, a diferencia de la mayoría de lenguajes de programación, no posee ninguna función nativa que facilite el redondeo de números decimales. Este pequeño inconveniente lo podemos suplir creando nosotros mismos una función que realice esta operación. Pero primero crearemos un programa muy sencillo para mostrar las ideas que utilizaremos.<br><br>`Import "mod_text";`<br>`import "mod_key";` |

```
process main()
private
 int num_redondeado;
 float num_a_redondear;
end
begin
  num_a_redondear=87.9;
  write(0,5,5,0, "Número a redondear:");
  write(0,5,15,0, "Número redondeado:");
  write_var(0,150,5,0, num_a_redondear);
  write_var(0,150,15,0, num_redondeado);
  loop
    if(num_a_redondear>=0)
      num_redondeado=num_a_redondear+0.5;
     end
    if(num_a_redondear<0)
      num_redondeado=num_a_redondear-0.5;
     end
    if(key(_up))
      num_a_redondear=num_a_redondear+0.01;
     end
    if(key(_down))
      num_a_redondear=num_a_redondear-0.01;
     end
    if(key(_esc)) break; end
    frame;
  end
end
```

Fíjate en el código: a la variable *num_a_redondear* se le asigna el valor que se quiere redondear. El programa mostrará por pantalla dicho valor, y además el resultado del redondeo. También, para comprobar que realmente el cálculo se hace correctamente, el programa ofrece la posibilidad de, mediante los cursores superior e inferior, subir o bajar el número a redondear -en este caso una centésima cada vez, pero puede ser cualquier otra cantidad, claro- , para poder ver dinámicamente que el resultado redondeado se amolda a los nuevos números.

Como se puede comprobar, el secreto del programa está simplemente en sumar o restar 0.5 al número a redondear. ¿Y esto por qué? Por un detalle muy importante:el numero a redondear es de tipo float, pero el número redondeado lo hemos definido como de tipo entero. Así que en realidad, quien realiza el redondeo es el propio Bennu, cuando hace la conversión de float a int: nosotros sólo le hemos dado el empujón. La necesidad de sumar o restar 0.5 viene del hecho que en realidad Bennu no redondea, sino corta por abajo: un 87.6 se convierte en un 87, un 109.8 se convierte en un 109, un 54.2 se convierte en un 54...Así que para conseguir el efecto de redondeo que todos conocemos (es decir, si las décimas son 5 o superior, se redondea al número inmediatamente superior) hay que hacer este pequeño truco.

A partir de aquí, es fácil crearnos nuestra propia función “round()” para poderla reutilizar en cualquier otro programa donde necesitemos esta funcionalidad. Esta función admitirá como parámetro el número a redondear y devolverá el número redondeado.

```
import "mod_text";
import "mod_key";

process main()
private
  int num_redondeado;
  float num_a_redondear=87.9;
end
begin
  write(0,5,5,0, "Número a redondear:");
  write(0,5,15,0, "Número redondeado:");
  write_var(0,150,5,0, num_a_redondear);
  write_var(0,150,15,0, num_redondeado);
  loop
    if(key(_up))
      num_a_redondear=num_a_redondear+0.01;
```

```
     end
    if(key(_down))
      num_a_redondear=num_a_redondear-0.01;
     end
    if(key(_enter)) num_redondeado=round(num_a_redondear); end
    if(key(_esc)) break; end
    frame;
  end
end

function round (float num_a_redondear)
private
    int num_redondeado;
end
begin
    if(num_a_redondear>=0)
      num_redondeado=num_a_redondear+0.5;
     end
    if(num_a_redondear<0)
      num_redondeado=num_a_redondear-0.5;
     end
    return num_redondeado;
end
```

Fíjate que la diferencia entre el código anterior y el primero que pusimos es que ahora sólo cuando se pulsa la tecla ENTER se ejecuta la función “round()” -la cual,fíjate, tiene un parámetro de tipo float pero devuelve un int-.

Este código no obstante tiene un inconveniente, y es que no podremos redondear números para una posición decimal concreta (no podremos redondear a décimas o centésimas o decenas): sólamente se puede redondear a números enteros. ¿Podríamos hacer que la función “round()” admitiera un segundo parámetro que fuera el nivel de redondeo deseado (hasta las décimas, centésimas, etc)? Pues sí; como muestra el siguiente código (lee los comentarios):

```
import "mod_text";
import "mod_key";
import "mod_math";   //Necesario para nuestra función round()
import "mod_string"; //Necesario para nuestra función round()
import "mod_regex";  //Necesario para nuestra función round()

Declare string round(float num_a_redondear, int decimales)
End

process main()
private
//  float num_redondeado;
  string num_redondeado;
  float num_a_redondear=87.967;
end
begin
  write(0,5,5,0, "Número a redondear:");
  write(0,5,15,0, "Número redondeado:");
  write_var(0,150,5,0, num_a_redondear);
  write_var(0,150,15,0, num_redondeado);
  loop
    if(key(_up))
      num_a_redondear=num_a_redondear+0.01;
    end
    if(key(_down))
      num_a_redondear=num_a_redondear-0.01;
    end
    if(key(_enter))
          while(key(_enter)) frame; end
          num_redondeado=round(num_a_redondear,3);
    end
    if(key(_esc)) break; end
    frame;
  end
```

```
end

function string round (float num_a_redondear, int decimales)
private
    int num_truncado;
    float num_redondeado;
    string cad_redondeada;
    string cadent;
    string caddec;
end
begin
/*Me aprovecho de que en Bennu, si se asigna un valor float a un entero, éste adquiere
el valor truncado de aquél. Por lo que multiplico el número a redondear por la potencia
de 10 necesaria para seguidamente asignarlo a una variable entera, la cual se quedará
sólo con la parte de la izquierda del punto decimal*/
    num_a_redondear=num_a_redondear*pow(10,decimales);
    if(num_a_redondear>=0)
      num_truncado=num_a_redondear+0.5;
     end
    if(num_a_redondear<0)
      num_truncado=num_a_redondear-0.5;
    end
/*Deshago la multiplicación anterior para volver al orden de magnitud original.La
división de int entre float (pow devuelve float) da float siempre, por lo que ahora no
perdemos ningún decimal en ningún truncamiento*/
    num_redondeado=num_truncado/pow(10,decimales);
/*Esta línea es una chapuza para evitar que en la división anterior num_redondeado se
quede por debajo del número que debería de ser (debido a la poca precisión de los
float) y entonces el redondeo se produzca a la baja. Para evitar esto, se le suma una
pequeña cantidad de un orden de magnitud menor que el límite de redondeo, para que éste
siempre quede mínimamente por encima del valor real de redondeo para cortarlo
adecuadamente.Es más fácil entenderlo si se comenta la línea y se observa qué pasa con
says.*/
    num_redondeado=num_redondeado + 5*pow(10,-(decimales+1));

/*Las líneas siguientes son para mostrar solamente el número de decimales deseado. El
resto de decimales que se obvian son totalmente aleatorios y espúreos y no sirven para
nada.*/
    //Convierto el float en cadena para poderla cortar
    cad_redondeada=ftoa(num_redondeado);
    //Primero obtengo la parte entera del número
    cadent=substr(cad_redondeada,0,regex("\.",cad_redondeada));
    //Ahora la decimal, pero sólo hasta donde se ha redondeado
    caddec=substr(cad_redondeada,regex("\.",cad_redondeada),decimales+1);
    //Y lo junto
    cad_redondeada=cadent+caddec;
    return cad_redondeada;
/*En vez de devolver una cadena, lo lógico sería que la función devolviera el número
float ya redondeado, pero si volvemos a hacer la conversión de cadena en float, resulta
que vuelven a aparecer los decimales espúreos más allá de donde hemos redondeado, hasta
completar un máximo de 6 decimales, los últimos sin sentido. Para observar este efecto,
se puede comentar la línea return anterior y descomentar las dos siguientes.*/
//    num_redondeado=atof(cad_redondeada);
//    return num_redondeado;
end
```

Comentar que en el código anterior se podría haber utilizado, de forma alternativa, la función **format()** para facilitar la tarea de cortar un número con un número de cifras determinadas. Se deja como ejercicio.

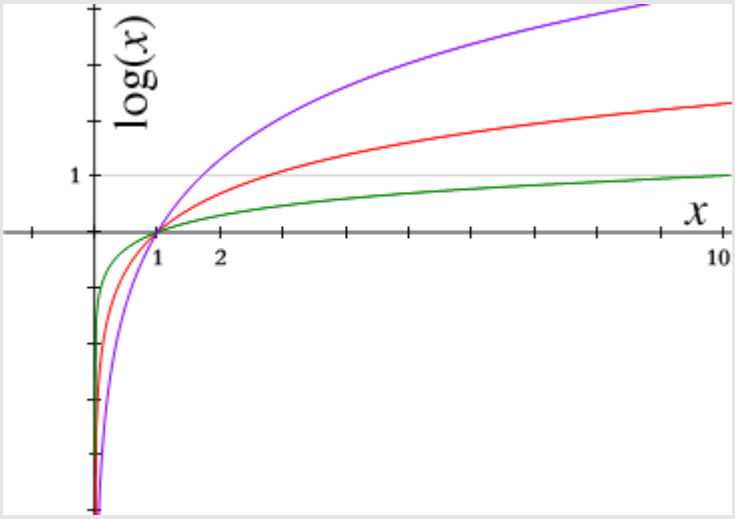
**Funciones logarítmicas**

Bennu también carece de funciones logarítmicas de cualquier base.

Recordemos que el logaritmo (con base *b*, -la cual ha de ser extrictamente positiva y diferente de 1-) de cualquier número *x* es el exponente *n* al que hay que elevar la base dada *b* para que nos dé dicho número *x*. Es decir:

$$\log_b x = n \Leftrightarrow \ x = b^n$$

Los tipos de logaritmos se clasifican según su base: los más importantes son los naturales -que tienen por base la constante matemática irracional *e*, cuyo valor es 2,718281828...-, los comunes -que tienen por base el número 10-, y tal vez los binarios -que tienen por base el número 2-. A continuación se muestra la representación de logaritmos en varias bases: el rojo representa el logaritmo en base *e*, el verde corresponde a la base 10 y el púrpura al de la base 1.7.




Estas funciones pueden sernos necesarias en determinados momentos (más adelante se verá un ejemplo de ello) y la fustración de no poderlas utilizar de forma nativa puede aparecer. No obstante, a grandes males, grandes remedios.

Es sabido en el ámbito del Cálculo Matemático que si una función F(x) cualquiera reúne una serie de requisitos (que afortunadamente las funciones logarítmicas cumplen ) se puede expresar como:

$$F(x) = \frac{F(a)}{0!} + \frac{F'(a)(x-a)}{1!} + \frac{F''(a)(x-a)^2}{2!} + \frac{F'''(a)(x-a)^3}{3!} + \frac{F^{(IV)}(a)(x-a)^4}{4!} + \ldots + \frac{F^{(n)}(a)(x-a)^n}{n!}$$

donde *a* puede ser cualquier valor real, $F^{(n)}(a)$ es la derivada n-ésima de F(x) con x=a y 0!=1 (el ! detrás de un número indica el factorial de ese número, por ejemplo: 4!=4·3·2·1).

Esto quiere decir en definitiva que podemos expresar F(x) como una suma de simples polinomios, y esto sí que Bennu lo puede calcular. Cuantos más polinomios de la suma se elijan para realizar la suma, mayor será la aproximación a la función exacta, por lo que hay que llegar a un compromiso entre un número razonable de elementos a sumar y el grado de exactitud que se requiere. De esta manera, podemos disponer de diferentes desarrollos de Taylor para las funciones que necesitemos, y que Bennu no tenga implementadas.

Por ejemplo, para hacer el desarrollo de Taylor de la función ln(x) (el logaritmo natural -también llamado neperiano-), tomando a=1 por comodidad, primero se calcularían las derivadas i-ésimas (las veces que necesitemos) en el punto x=1 y se irían obteniendo los diferentes elementos de la suma de polinomios. El resultado de todo ello lo presento a continuación: el desarrollo más útil del logaritmo natural de cualquier número real mayor que 0 (de hecho, ln(x) no está definida para x<=0) es:

$$\textbf{Si hacemos } y=(x-1)/(x+1)\textbf{ , entonces: } \quad \ln(x) = 2 \cdot \{ y + y^3/3 + y^5/5 + \ldots \}$$

Para realizar el cálculo del logaritmo decimal de cualquier número (o logaritmo de cualquier otra base), se puede aplicar la conocida regla de los logaritmos para el cambio de base, sabiendo antes su logaritmo natural . Por ejemplo, para el logaritmo decimal de un número x:

$$\log_a(x) = \log_b(x) / \log_b(a) \quad \xrightarrow{\text{si hacemos a=10 y b=}e} \quad \log_{10}(x) = \ln(x) / \ln(10) = \ln(x) / 2.302585$$

Otros desarrollos interesantes (consultar por ejemplo la Wikipedia para más información) podrían ser las funciones

hiperbólicas o la serie geométrica.

Así pues, podremos utilizar en nuestros programas las funciones logarítmicas (natural y decimal) sin más que implementar nosotros mismos su código a partir de sus respectivos desarrollos. Dicho y hecho:

```
import "mod_text";
import "mod_key";
import "mod_video";
import "mod_math"; //Requerida por nuestras propias funciones ln() y log()

Declare float ln(float numero)
End
Declare float log(float numero)
End

process main()
private
  float numero=10.6;
end
begin
  set_mode(500,50);
  write(0,0,0,0,"El logaritmo natural de " + numero + " es: " + ln(numero));
  write(0,0,10,0,"El logaritmo decimal de " + numero + " es: " + log(numero));
  write(0,0,30,0,"El exponencial de " + numero + " es: " + pow(2.718281,numero));
  repeat
    frame;
  until(key(_esc))
end

/*A pesar de los 30 términos incluidos en el valor devuelto, a partir aproximadamente
del número 20 dado como entrada, se pierde la precisión en las milésimas, así que
cuidado. Se podría modificar esta función para que en vez de tener un número fijo de
términos en el desarrollo de Taylor, este número se pudiera establecer mediante un
segundo parámetro, en cuyo caso el desarrollo se construiría a partir de un bucle que
iría añadiendo los elementos que se hayan especificado. Se deja como ejercicio.*/
function float ln(float numero)
private
  float aux;
end
begin
  aux=(numero-1)/(numero+1);
  return  ( 2*(aux  +  pow(aux,3)/3  +  pow(aux,5)/5  +  pow(aux,7)/7  +  pow(aux,9)/9  +
pow(aux,11)/11  +  pow(aux,13)/13  +  pow(aux,15)/15  +  pow(aux,17)/17  +  pow(aux,19)/19  +
pow(aux,21)/21  +  pow(aux,23)/23  +  pow(aux,25)/25  +  pow(aux,27)/27  +  pow(aux,29)/29  +
pow(aux,31)/31  +  pow(aux,33)/33  +  pow(aux,35)/35  +  pow(aux,37)/37  +  pow(aux,39)/39  +
pow(aux,41)/41  +  pow(aux,43)/43  +  pow(aux,45)/45  +  pow(aux,47)/47  +  pow(aux,49)/49  +
pow(aux,51)/51 + pow(aux,53)/53 + pow(aux,55)/55 + pow(aux,57)/57) );
end

//La precisión viene determinada por la exactitud de la función ln()
function float log(float numero)
begin
  return (ln(numero)/2.302585);
end
```

Posiblemente eches de menos la función exponencial ($f(x)=e^x$ )en Bennu también. Se podría implementar también utilizando su desarrollo de Taylor, pero en verdad no es necesario. Fíjate que no es más que una potencia cuya base es el número *e*. Por lo tanto, con una simple línea similar a *pow(2.71828, x);* ya tendremos nuestra función exponencial lista para ser usada.

**ISINF(NUMERO)**

Esta función, definida en el módulo "mod_math", devuelve 1 (TRUE) si el número o expresión numérica pasado por parámetro tiene un valor de +/- infinito, y 0 (FALSE) en caso contrario.

Esta función se puede utilizar para detectar de forma precoz posibles problemas en el cálculo de diversas operaciones, como por ejemplo, la división por 0.

PARÁMETROS: FLOAT NÚMERO : Número en coma flotante

**ISNAN(NUMERO)**

Esta función, definida en el módulo "mod_math", devuelve 1 (TRUE) si el parámetro o expresión pasado por parámetro tiene un valor numérico indeterminado , y 0 (FALSE) si dicho parámetro contiene una valor numérico concreto. Las indeterminaciones matemáticas pueden ser de muchos tipos: división de 0 entre 0, división de infinito entre infinito, raíz cuadrada de un número negativo, etc.

Esta función se puede utilizar para detectar de forma precoz posibles problemas problemas en el cálculo de diversas operaciones donde aparezca indeterminaciones matemáticas.

PARÁMETROS: FLOAT NÚMERO : Número en coma flotante

**FINITE(NUMERO)**

Esta función, definida en el módulo "mod_math", devuelve lo contrario que **isinf()**: devuelve 0 (FALSE) si el número o expresión numérica pasado por parámetro tiene un valor de +/- infinito, y 1 (TRUE) en caso contrario. La única circunstancia donde ambas funciones devolverán el mismo valor es cuando se produzca una indeterminación matemática (0/0, inf/inf, etc).

PARÁMETROS: FLOAT NÚMERO : Número en coma flotante

Un ejemplo de las tres funciones anteriores podría ser el siguiente:

```
import "mod_say";
import "mod_math";

Process Main()
Private
    float n = 10,n1 = 0;
End
Begin
//El resultado de la operación devuelve error #INF , porque es una división por 0
    say ("n=" + n + " n1=" + n1 );
    say ("n/n1=" + n/n1 );
    say ("isinf(n/n1)="+ isinf(n/n1) );
    say ("isnan(n/n1)="+ isnan(n/n1) );
    say ("finite(n/n1)="+ finite(n/n1));
    say ("");
    say ("n=" + n);
    say ("n/2=" + n/2 );
    say ("isinf(n/2)="+ isinf(n/2) );
    say ("isnan(n/2)="+ isnan(n/2) );
    say ("finite(n/2)="+ finite(n/2));
    say ("");
    say ("n=" + n);
    say ("(1+n/0.0)=" + (1+n/0.0) );
    say ("isinf((1+n/0.0))="+ isinf((1+n/0.0)));
    say ("isnan((1+n/0.0))="+ isnan((1+n/0.0)));
    say ("finite((1+n/0.0))="+ finite((1+n/0.0)));
//El resultado de la operación devuelve error #IND, porque es una indeterminación (0/0,inf/inf,etc)
    say ("");
    say ("0.0/0.0=" + 0.0/0.0 );
    say ("isinf(0.0/0.0)="+ isinf(0.0/0.0));
    say ("isnan(0.0/0.0)="+ isnan(0.0/0.0));
    say ("finite(0.0/0.0)="+ finite(0.0/0.0));
    say ("");
    n = -1;
    say ("n=" + n );
```

```
    say ("sqrt(n)=" + sqrt(n));
    say ("isinf(sqrt(n))="+ isinf(sqrt(n)));
    say ("isnan(sqrt(n))="+ isnan(sqrt(n)));
    say ("finite(sqrt(n))="+ finite(sqrt(n)));
/*Aquí no hay ningún error, ya que aunque n sea infinita, los parámetros de las funciones
vale 0/inf, es decir 0,un número perfectamente válido*/
    say ("");
    n = 1/0.0;
    say ("n=" + n + " (1/0.0)");
    say ("1.0/n=" + 1.0/n );
    say ("isinf(1.0/n)="+ isinf(1.0/n));
    say ("isnan(1.0/n)="+ isnan(1.0/n));
    say ("finite(1.0/n)="+ finite(1.0/n));
/*Se comprueba los límites máximos y mínimos de un float, para comprobar el
funcionamiento correcto de las funciones*/
    say ("");
    n =
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000; //130 zeros
    say ("n=" + n + "
(1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000)");
    say( "isinf(n)=" + isinf(n) );
    say( "isnan(n)=" + isnan(n) );
    say( "finite(n)=" + finite(n) );
    say ("");
    n =
0.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000001; //130 zeros
    say ("n=" + n + "
(0.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000001)");
    say( "isinf(n)=" + isinf(n) );
    say( "isnan(n)=" + isnan(n) );
    say( "finite(n)=" + finite(n) );
End
```

Los números aleatorios son ampliamente utilizados en todo tipo de videojuegos. Ya conocemos la función **rand()**, pero existe otra función muy importante que tiene que ver con los números aleatorios: **rand_seed()**.

| **RAND_SEED(NUMERO)** |
|---|
| Sabemos de un capítulos anteriores que para generar números pseudoaleatorios disponemos de la función **rand()**. ¿Por qué se llaman números pseudoaleatorios y no aleatorios? Porque tal como funciona **rand(),** los números que genera no son completamente independientes unos de otros, y por tanto, no son exactamente aleatorios. Que una serie de números esté formada por números exactamente aleatorios implica que no se pueden hallar patrones de números que se repitan periódicamente; los números generados por **rand()** realmente sí cumplen un patrón (es difícil de detectar, pero existe).<br><br>La función **rand()** para funcionar lo que utiliza es un número inicial, llamado "semilla" (seed), el cual será el generador de toda la ristra de números pseudoaleatorios a obtener. Es decir, la función **rand()** necesita siempre de ese número para poder empezar a trabajar. Y lo importante del tema está en que si se escoge como semilla un mismo número siempre, la ristra de números ¡siempre será la misma!, porque a partir de la semilla se genera el primer número; a partir del primero se genera el segundo, y así. Claro, si la semilla es la misma, la ristra también. Para evitar esto, y hacer que cada vez que un programa que incluya la función **rand()** genere ristras de números completamente diferentes, Bennu hace que automáticamente el valor tomado como semilla de la función **rand()** sea la hora (en segundos) actual. Evidentemente, este número cada segundo cambiará, por lo que todos los programas que ejecutemos en segundos diferentes tendrán ristras de números diferentes.<br><br>No obstante, a veces nos puede ser de utilizar conseguir precisamente lo contrario: es decir, hacer que cada vez que se ejecute un programa genere siempre la misma ristra de valores. Para hacer eso, deberíamos poder fijar el valor de la semilla. Y eso es lo que hace la función **rand_seed()**, definida en el módulo "mod_rand".. El parámetro que tiene es un entero que será la semilla a partir de la cual se generará la serie de números pseudoaleatorios. Y si hacemos que ese parámetro siempre valga lo mismo, ya tendremos asegurada una serie siempre igual. |

Veámoslo con un ejemplo. Ejecuta el siguiente programa.

```
import "mod_file";
import "mod_rand";
import "mod_timers";
import "mod_string";


process main()
private
          int mirand;
          int mifich;
end
begin
          if(file_exists("prueba.txt")==false)
                    mifich=fopen("prueba.txt",o_write);
          else
                    mifich=fopen("prueba.txt",o_readwrite);
                    fseek(mifich,0,2);
          end
          while(timer[0]<=500)
                    mirand=rand(1,9);
                    fputs(mifich,itoa(mirand));
                    frame;
          end
          fclose(mifich);
end
```

Este programa lo que hace es crear un fichero si no existe (o utilizar uno ya existente) y escribir en él –sin sobreescribir nada si hubiera algo ya- una serie de valores pseudoaleatorios, durante 5 segundos.

Ejecútalo una vez y abre el fichero de texto generado. Verás que es una serie de números sin ningún sentido. Ve al final del fichero y escribe un carácter que no sea un número. Éste será la marca que indicará el comienzo de una serie nueva. Vuelve a ejecutar el programa, y vuelve a abrir el fichero. Verás que justo después de la marca que habías escrito vuelve a aparecer una serie de números, que no tienen nada que ver con los de la serie anterior. Esto es porque **rand()** automáticamente coge la semilla de la hora del sistema, y la hora del sistema ha cambiado entre las dos ejecuciones del programa.

Ahora modifica el código anterior añadiendo la línea siguiente justo después del begin:

```
rand_seed(3);
```

Borra el archivo de texto y vuelve a realizar los pasos anteriores: ejecutar una vez el programa, escribir una marca al final del fichero y volver a ejecutar el programa. Verás que ahora, como hemos definido en el programa que la semilla a utilizar cuando se ejecute **rand()** sea siempre la misma (en este caso, el número 3), las dos series de números, pese a ser pseudoaleatorias, ¡son las mismas! Por otro lado, podrás deducir conmigo fácilmente que poner *rand_seed(time());* y no poner nada sería lo mismo, pues.

Finalmente, como colofón a este apartado final dedicado a la aleatoriedad de los números,, a continuación realizaremos una demostración gráfica de cuánto de aleatorio tienen realmente los números generados por **rand()**. El código siguiente genera números aleatorios sin parar, y muestra una serie de barras (de la número 0 hasta la 48), cuyo tamaño horizontal (y color) es proporcional al número de veces que el número correspondiente a esa barra ha sido generado por **rand()**. Lo ideal sería que ninguna barra fuera más larga que la otra, cosa que ocurre si dejamos al programa ejecutarse una cierta cantidad de tiempo (teniendo en cuenta las típicas fluctuaciones que no se pueden corregir en un tiempo finito). Compruébalo:

```
import "mod_video";
import "mod_map";
import "mod_text";
import "mod_rand";
import "mod_key";
import "mod_time";
import "mod_string";
```

```
import "mod_math";

global
          int barra_verde;
          int barra_roja;
          int barra_azul;
          int maxvalue=1;
          int acum[49];
end

process main()
Private
  int i;
  int n=0;
end
Begin
set_mode(640,480);
rand_seed(time()*10);

//Genero los gráficos de las barras (verdes,rojas ó azules)
barra_verde=new_map(640,10,32);
map_clear(0, barra_verde, rgb(0,255,0));
barra_roja=new_map(640,10,32);
map_clear(0, barra_roja, rgb(255,0,0));
barra_azul=new_map(640,10,16);
map_clear(0, barra_azul, rgb(0,0,255));

/*Imprimo la lista de números, y la cantidad de veces que se ha ido obteniendo cada uno
de rand() en sucesivas ocasiones*/
from i=0 to 48;
          write (0, 10,i*10,0,itoa(i)+":");
          write_var(0, 50,i*10,0,acum[i]);
          barra(i,0,i*10);
end

//Genero números aleatorios sin parar
while(!key(_ESC))
          n = rand(0,48);
          acum[n]++;
/*Guardo siempre en maxvalue la cantidad máxima de veces que se ha obtenido algún número.
Esto es para luego dibujar de forma proporcional la longitud de la barra*/
          if (acum[n]>maxvalue)
                    maxvalue=acum[n];
          end
          Frame;
End
End

process barra(i,x,y)
begin
graph=barra_roja;
// write_var(0, 100,i*10,0,size_x); // Porcentaje
while(!key(_ESC))
/*El  tamaño  horizontal  de  la  barra  estará  siempre  acorde  a  la  proporción
cantidadvecesquehaaparecidoelnumero/cantidadmaxima,  de  manera  que  se  mantenga  dicho
tamaño siempre por debajo de la anchura de la pantalla*/
          size_x=acum[i]*100/maxvalue;
//Y además, dependiendo de dicho tamaño horizontal,la barra será de un color u otro
          switch(size_x)
                    case 0..33:
                              graph=barra_verde;
                    end
                    case 34..66:
                              graph=barra_azul;
                    end
                    case 67..100:
                              graph=barra_roja;
                    end
```

```
            end
            frame;
end
end
```

¿Cómo podríamos hacer ahora, por ejemplo, que los valores centrales fueran más probables y los de los extremos menos? Es decir, ¿cómo podríamos hacer que los números no fueran tan aleatorios y que los del centro tuvieran más posibilidad de aparecer? Un truco podría ser pensar por ejemplo en tener dos dados en vez de uno: si tenemos un dado, todos los números del 1 al 6 tienen la misma probabilidad, pero si tenemos dos, las combinaciones posibles hacen que los números centrales aparezcan más que los de los extremos. Por ejemplo, sólo existe una combinación posible para que salga un 1 doble -(1,1)- y sólo dos para que salga un 11 -(5,6), (6,5)- pero cinco para que salga un 6 -(1,5),(5,1),(2,4),(4,2),(3,3)-y seis para que salga un 7 -(1,6),(6,1),(2,5),(5,2),(3,4),(4,3)-. Podemos hacer lo mismo con el programa anterior: en vez de hacer un **rand()** del 0 al 48, podemos hacer dos **rand()** de 0 a 24 cada uno y sumarlos: así habrá más probabilidades de que los números alrededor del 24 aparezcan más a menudo. Por lo tanto, sustituye la línea donde aparece **rand()** en el código anterior y escribe:

```
n = rand(0,24) + rand(0,24);  //Más frecuentes los valores intermedios
```

Otra manera de conseguir el mismo objetivo podría ser, si la cantidad de valores es relativamente pequeña, almacenar en un array todos los diferentes valores, algunos de ellos repetidos. Por ejemplo *int r[] ={1,2, 2,3,3,3,4,4,5}* Si luego se escoge un elemento de dicho array dado por la expresión *r[rand(0, sizeof(r)- 1)]* (es decir, se elige un elemento al azar), dependiendo del número de veces que esté ese valor repetido en el array, será más probable o menos que salga escogido.

También podríamos hacer que los valores más probables fueran los más elevados, modificando la línea del **rand()** del código anterior por ésta otra:

```
n = 48 - abs(rand(0,48) - rand(0,24)); //Más frecuentes los valores altos
```

O al contrario, los valores más inferiores:

```
n = abs(rand(0,48) - rand(0,24));  //Más frecuentes los valores bajos
```

Otra manera de conseguir esto último sería utilizando funciones logarítmicas, de la siguiente manera: *n = pow(10,rand(0,log(50)));* donde la el primer parámetro de **pow()** debería ser la base del logaritmo a utilizar (de hecho, cuanto menor sea éste número, mayor diferencia habrá entre el número de veces que aparecen los números pequeños y los grandes). No obstante, la fórmula anterior sólo funciona para nuestros propósitos si la función **rand()** acepta como parámetros valores decimales y devuelve también valores decimales, cosa que en Bennu no ocurre, por lo que esta opción no la podremos utilizar.

## *Trabajar con el ratón

El ratón no se programa como un proceso normal: su funcionamiento se basa en una estructura, la estructura global **"mouse"**, definida en el módulo "mod_mouse". Ésta dispone de una serie de campos que se pueden consultar y a los que se les puede asignar valores, no necesitando ningún proceso.

Por ejemplo, los campos **"x"** e **"y"** de la estructura **"mouse"** (es decir, **"mouse.x"** y **"mouse.y"**) se utilizan para consultar y asignar la posición del cursor del ratón. Sus coordenadas son respecto a la ventana y sus valores máximos dependen del modo de pantalla (800x600, 640x480,...). También puedes ponerle al cursor del ratón el dibujo del puntero que quieras: sólo tienes que asignarle los valores adecuados a la variable **"mouse.file"** y **"mouse.graph"**. También puedes cambiarle el tamaño, como si del gráfico de un proceso se tratara, con **"mouse.size"**. Y sus flags, al igual que cualquier otro gráfico, con **"mouse.flags"** (con los mismo valores posibles). Y su orientación con **"mouse.angle"**. Y su profundidad con **"mouse.z".** También podemos especificar la región de la pantalla donde queremos que sólo allí sea visible el puntero, con **"mouse.region"** -el tema de las regiones se tratará en este capítulo más adelante-.Para comprobar la pulsación de los botones puedes comprobar, como si fueran teclas del teclado, las variables **"mouse.left"**, **"mouse.right"** y **"mouse.center"** para ver si se ha pulsado el botón izquierdo, derecho o central respectivamente. Puedes hacer uso también de la función *collision* de la forma **"collision (type mouse);"** para comprobar si el gráfico de un proceso ha hecho contacto con el "punto caliente" del cursor (el punto de control 0, que

normalmente es la punta de la flecha que selecciona el objeto), algo muy útil para hacer que cambie éste con el contacto.

Como ves, tienes un control completo sobre el ratón y con un manejo sencillísimo.Un ejemplo:

```
import "mod_text";
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_grproc";

process main()
begin
          proceso();
          mouse.file=0;
          mouse.graph=new_map(20,20,32);map_clear(0,mouse.graph,rgb(205,24,46));
          write_var(0,100,20,4,mouse.x);
          write_var(0,100,40,4,mouse.y);
          write_var(0,100,60,4,mouse.angle);
          write_var(0,100,80,4,mouse.size);
          write_var(0,100,100,4,mouse.flags);
          write_var(0,100,120,4,mouse.z);
          write_var(0,100,140,4,mouse.region);
          loop
//Por si se quiere mover el ratón con el teclado
            if(key(_x)) mouse.x=mouse.x+5;end
            if(key(_y)) mouse.y=mouse.y+5;end
//Gira la orientación del gráfico del puntero
            if(key(_a)) mouse.angle=(mouse.angle+5000)%360000;end
//Aumenta el tamaño del gráfico del puntero
            if(key(_s)) mouse.size=mouse.size+10; end
/*El while está puesto para que mientras se tenga pulsada la tecla no se haga nada, y
sólo se cambie el flag una vez se ha soltado. Esto es para evitar que, si se mantiene
pulsada más de la cuenta la tecla, el flag varíe demasiados valores de golpe*/
            if(key(_f)) while(key(_f))frame;end mouse.flags=(mouse.flags+2)%16;end
//Aun teniendo Z>0, no se pinta por debajo del proceso (!!!?)
            if(key(_z)) mouse.z++; end
/*Si la región deja de ser la nº0, que representa la pantalla completa, el puntero deja
de ser visible*/
            if(key(_r)) mouse.region++;end
            if(mouse.left==true) write(0,100,160,4,"Botón izquierdo pulsado");end
            if(mouse.right==true)  write(0,100,180,4,"Botón derecho pulsado");end
            if(key(_esc)) exit(); end
            frame;
          end
end

Process proceso()
private
          int a;
end
begin
          graph=new_map(50,50,32);map_clear(0,graph,rgb(255,255,0));
          x = 160;
          y = 120;
          loop
               if(key(_up))y=y-5;end
               if(key(_down))y=y+5; end
               if(key(_left))x=x-5; end
               if(key(_right))x=x+5; end
               if(collision(type mouse)) write(0,100,200,4,"¡Colisión!");end
       frame;
   end
end
```

No obstante, suele ser habitual que en vez de utilizar la estructura ***MOUSE*** “a pelo”, se utilice para

controlar el puntero del ratón un proceso normal y corriente que lo único que haga sea actualizar permanentemente la posición de su gráfico a la del puntero (invisible en este caso). Con esta manera de hacer, se gana flexibilidad y versatilidad (sobretodo en la detección de colisiones) como veremos en los ejemplos siguientes. Cabe destacar también que de este modo, los únicos campos que se utilizarán de la estructura ***MOUSE*** son **"mouse.x"** y **"mouse.y"**, para asignar su valor en cada iteración a la ***X*** e ***Y*** del proceso en cuestión, pero ni **"mouse.file"**, ni **"mouse.graph"**, ni **"mouse.size"**, ni **"mouse.angle"** ni **"mouse.flags"** ni **"mouse.z"** ni **"mouse.region"** tendrán utilidad, ya que en su lugar usaremos las correspondientes variables locales para ese proceso en cuestión (***FILE,GRAPH***,etc). Como muestra,a continuación pongo el mismo ejemplo anterior pero haciendo que el puntero del ratón sea un proceso "estándar".

```
import "mod_text";
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_grproc";

process main()
begin
          proceso();
          raton();
/*Podemos ver que si modificamos el ANGLE,SIZE,etc del proceso "raton" -y observamos su
efecto en el gráfico del puntero- los valores de la estructura mouse (cuyo gráfico de
puntero es invisible) no se ven modificados para nada*/
          write_var(0,100,60,4,mouse.angle);
          write_var(0,100,80,4,mouse.size);
          write_var(0,100,100,4,mouse.flags);
          write_var(0,100,120,4,mouse.z);
          write_var(0,100,140,4,mouse.region);
          loop
            if(mouse.left==true) write(0,100,160,4,"Botón izquierdo pulsado");end
            if(mouse.right==true)  write(0,100,180,4,"Botón derecho pulsado");end
            if(key(_esc)) exit(); end
            frame;
          end
end

process raton()
begin
  file=0;
  graph=new_map(20,20,32);map_clear(0,graph,rgb(205,24,46));
  loop
/*Estas dos líneas siguientes simularán que este proceso es el puntero de ratón: asigna
la posición del puntero real (invisible) a la posición del proceso, cuyo gráfico será el
que queramos. IMPORTANTE ponerlas dentro de un bucle, para ir actualizando las
coordenadas a cada frame.*/
            x=mouse.x;
            y=mouse.y;
            if(key(_x)) x=x+5;end
            if(key(_y)) y=y+5;end
            if(key(_a)) angle=(angle+5000)%360000;end
            if(key(_s)) size=size+10; end
            if(key(_f)) while(key(_f))frame;end flags=(flags+2)%16;end
            if(key(_z)) z++; end
            if(key(_r)) region++;end
            frame;
   end
end

Process proceso()
private
          int a;
end
begin
           graph=new_map(50,50,32);map_clear(0,graph,rgb(255,255,0));
          x = 160;
          y = 120;
```

```
          loop
               if(key(_up))y=y-5;end
               if(key(_down))y=y+5; end
               if(key(_left))x=x-5; end
               if(key(_right))x=x+5; end
//Ahora detecto la colisión con el proceso "ratón", no con la estructura MOUSE
               if(collision(type raton)) write(0,100,200,4,"¡Colisión!");end
                frame;
           end
end
```

A partir de aquí, se nos pueden ocurrir muchas aplicaciones del cursor del ratón en nuestros juegos. Por ejemplo,¿cómo podríamos hacer por ejemplo que un proceso cualquiera estuviera mirando constantemente al movimiento del ratón?. Pues una manera sería:

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_grproc";

process main()
begin
          observador();
          destino();
          repeat
                    frame;
          until(key(_esc))
          let_me_alone();
end

process observador()
private
          int angulo;
end
begin
          x=160;y=120;
          graph = new_map(50,50,32);map_clear(0,graph,rgb(255,255,0));
          loop
                    angle=get_angle(get_id(type destino));
                    frame;
          end
end

process destino()
begin
          graph = new_map(10,10,32);map_clear(0,graph,rgb(255,0,255));
          loop
                    x=mouse.x;
                    y=mouse.y;
                    frame;
          end
end
```

No creo necesario explicar nada: espero que se entienda todo: **get_angle()** devuelve el ángulo entre la horizontal y el proceso observado cuyo identificador se le pasa como parámetro (identificador, por cierto, que en este ejemplo lo obtenemos gracias a **get_id()**). Y el ***ANGLE*** de cualquier proceso sabemos que es el ángulo entre la línea horizontal y donde está apuntando en este momento. Por lo tanto:al igualar ambos valores estamos diciendo que el ángulo respecto a la línea horizontal al que ha de apuntar el proceso observador sea el mismo que el ángulo respecto a la línea horizontal donde está el proceso "destino()": es decir, que el proceso "observador()" apunte a "destino()". A lo mejor se te hubiera ocurrido poner *angle=get_angle(mouse);*. Sin embargo, no habría funcionado porque ***MOUSE*** no es un proceso,es una estructura.

Una alternativa a la ya escrita hubiera sido sustituir la línea *angle=get_angle(get_id(type destino));* por ésta otra: *angle=fget_angle(x,y,mouse.x,mouse.y);*. Recuerda que **fget_angle()** básicamente lo que hace es calcular el ángulo (en milésimas de grado) formado por una línea determinada y la línea horizontal (el eje X). Los cuatro

parámetros que tiene son respectivamente la coordenada X del primer punto de la línea, la coordenada Y del primer punto de esa línea, la coordenada X del segundo punto de esa línea y la coordenada Y del segundo punto de esa línea (una línea se forma por la unión de dos puntos, claro).

Bueno, volviendo a lo que teníamos entre manos, fíjate como cosa curiosa que con el proceso "destino()" tenemos una manera de cambiar el icono del ratón.

Cambiemos de tema. Otro truco muy útil es hacer que el gráfico de un proceso se dirija allí donde se haya hecho clic con el ratón. ¿Cómo se podría hacer esto? Imaginemos que tenemos un proceso "personaje()". Lo que necesitaremos hacer es, primero, crear dos variables globales, por ejemplo *posx* y *posy*, y seguidamente, escribir en el bucle Loop principal del proceso "personaje()" algo así (es fácil de entender):

```
loop
          if(mouse.left==true)
                    posx=mouse.x;
                    posy=mouse.y;
                    mouse.left=false;
          end
          if(x>posx); x=x-1; end
          if(x<posx); x=x+1; end
          if(y>posy); y=y-1; end
          if(y<posy); y=y+1; end
          frame;
end
```

A continuación pongo un código muy curioso, que nos permite dibujar en la pantalla con el puntero del ratón igual que si utilizáramos alguna herramienta de dibujo a mano alzada, tipo lápiz.

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";

process main()
Begin
  definemouse();
  while (not key(_esc))
    frame;
  end
  exit();
End

Process definemouse()
Begin
  mouse.x=160;
  mouse.y=120;
  graph=new_map(20,20,32);map_clear(0,graph,rgb(255,0,0));
Loop
    If (mouse.x>320) mouse.x=320;  End
    If (mouse.x<0)   mouse.x=0;    End
    If (mouse.y>240) mouse.y=240;  End
    If (mouse.y<0)   mouse.y=0;    End
    If (mouse.left)  dibuja(mouse.x,mouse.y); End
    x=mouse.x;
    y=mouse.y;
    Frame;
  End
End

Process dibuja(x,y)
Begin
  flags=4;
  graph=new_map(20,20,32);map_clear(0,graph,rgb(255,0,0));
  Loop
    Frame;
  End
```

```
End
```

El siguiente ejemplo es un mero efecto estético, similar al que visto anteriormente que simulaba un pincel, pero con un comportamiento diferente, más sofisticado y espectacular. Este efecto es ideal para simular gases de propulsión, lanzallamas,etc.

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_rand";

global
          int idpng;
end

process main()
begin
idpng=new_map(20,20,32);map_clear(0,idpng,rgb(0,0,255));
mouse.graph=idpng;
loop
          if (mouse.right) llama(mouse.x,mouse.y); end
          if (key(_esc)) exit(); end
          frame;
end
end

process llama(x,y)
private
          byte i;
end
begin
graph=idpng;
size=rand(5,20);
angle=rand(0,360)*1000;
flags=4;
repeat
          x=x+5;
          z++;
          angle=(angle+5000)%360000;
          i=i+5;
          frame;
          size=size+7;
until(i>110)
end
```

Otro código interesante: lo que hace es alterar la sensibilidad del ratón con un par de sencillos procesos que podemos trasladar a cualquiera de nuestros programas. De esta manera, podremos hacer que el cursor del ratón vaya más ligero o que le cueste moverse (como si pesara toneladas), ofreciendo así un efecto muy resultón:

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_draw";

const
   SENSIBILITY_FAST=1;
   SENSIBILITY_SLOW=5;
end

global
/*Es float por si queremos hacer sensibilidades decimales,o menores de 1 para hacer el
cursor ultrarápido*/
   float sensibility;
end
```

```
process main()
begin
  sensibility= SENSIBILITY_FAST; //Sensibilidad inicial
  write(0,10,10,0,"Pulsa (R)ápido    (L)ento");
  mouse_test();
  loop
     if (key(_R))  sensibility= SENSIBILITY_FAST;  end
     if (key(_L))  sensibility= SENSIBILITY_SLOW;  end
     if (key(_esc)) exit(); end
     frame;
  end
end

process mouse_test()
begin
//Este proceso representa el puntero del ratón
x = mouse.x;
y = mouse.y;
graph = create_mouse_pointer();
loop
/*Lo importante del ejemplo: la nueva posición del gráfico del proceso vendrá dada por el
valor de "sensibility". Se puede ver que si ésta vale 1 (rápida), x=x+(mouse.x-x)/1 ->
x=mouse.x; pero si la sensibilidad vale 5 (lenta), x=x+(mouse.x-x)/5 -> x=5*mouse.x +
4*x/5. Se pueden buscar otras fórmulas para simular el mismo efecto.*/
     x = x +(mouse.x-x)/(sensibility);
     y = y +(mouse.y-y)/(sensibility);
     frame;
end
end

function create_mouse_pointer()
private
  int copo;
end
begin
copo=new_map(3,3,32);
/*Todas las primitivas gráficas que se utilicen a partir de ahora,si no se dice lo
contrario, se van a pintar sobre la imagen acabadade generar -inicialmente vacía de
contenido gráfico-. Esto es una manera de decir que se rellenará de contenido gráfico
esta imagen con las primitivas gráficas que usemos a partir de ahora.*/
drawing_map(0,copo);
//Dibujamos un "copo de nieve" sobre la imagen "copo"
drawing_color(RGB(250,250,250));
draw_line(1,1,3,3);
draw_line(2,1,2,3);
drawing_color(RGB(255,255,255));
draw_line(1,2,3,2);
draw_line(3,1,1,3);
/*Devolvemos el identificador de la imagen creada con new_map, (y rellenada de contenido
gráfico con las primitivas sucesivas). Este identificador devuelto puede ser utilizado
para cualquier cosa que necesite asignarse a un código de imagen, como por ejemplo, en
este caso, la variable global GRAPH. De esta manera, la imagen del proceso mouse_test
será la correspondiente al identificador devuelto, que es el "copo de nieve".*/
return copo;
end
```

El siguiente ejemplo es una implementación para detectar un doble click. El código muestra un cuadrado en el centro de la pantalla, sobre el cual se podrá hacer click con el cursor del ratón (y fuera del cuadrado también,claro), pero solamente cuando se haga doble click dentro del cuadrado, se notificará con un mensaje. Si se hace doble click fuera de él, o clicks simples tanto dentro como fuera, no pasará nada.

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_text";
```

```
import "mod_draw";
import "mod_grproc";

process main()
begin
   process_mouse();
   caja_comprobacion();
end

process process_mouse()
begin
    mouse.graph = new_map(20,20,32);map_clear(0,mouse.graph,rgb(100,100,200));
    mouse.x=160;mouse.y=120;
    loop
       if (key(_esc)) exit(); end
       frame;
    end
end

process caja_comprobacion()
private
//Vale 0 si no se ha hecho ningún click, 1 si se acaba de hacer uno y 2 si se acaba de
hacer dos
    int clickeo;
//Tiempo entre un primer click y el segundo
     int stop;
//Tiempo después del segundo click
     int time;
end
begin
      graph = new_map(50,50,32);map_clear(0,graph,rgb(255,0,255));
      x = 160;y = 120;
      loop
//Si es la primera vez que se clica sobre el proceso...
          if ((collision(type mouse) AND (mouse.left) AND (clickeo==0))) clickeo = 1; end
/*A partir de hacer un click, se aumenta a cada frame "stop". Si "stop" supera el valor 8
(es decir, si pasan ocho frames -esto se puede regular- ), se resetea al estado inicial*/
          if (clickeo==1) stop++; end
          if (stop>=8) stop=0; clickeo = 0; time=0; end
/*Si es la segunda vez que se clica sobre el proceso...(y ese segundo click se ha hecho
relativamente separado del primero -porque "stop" ha de valer más de 4 y no valdrá nunca
más de 8...-) */
           if ((collision(type mouse) AND (mouse.left) AND (clickeo==1) AND (clickeo==1)
AND (stop>=4)))
               clickeo = 2;
           end
/*Si se ha hecho doble click, se nofifica y se aumenta a cada frame "time". Si "time"
supera el valor 14 (es decir, si pasan 14 frames -esto se puede regular- ), se borra la
notificación y se resetea al estado inicial*/
           if ((clickeo==2)) write(0,0,0,0,"Segundo clickeo"); time++; end
           if (time>=14) delete_text(0); clickeo=0; time=0; stop=0; end
           frame;
      end
end
```

Finalmente, no está de más comentar la existencia de la variable global ***MOUSE_STATUS***, la cual vale 1 cuando el puntero del ratón está dentro de los límites de la ventana (si la aplicación se ejecuta en modo ventana) y 0 cuando sale de ella. Este valor no tiene utilidad si la aplicación se ejecuta en modo de ventana modal (usando el parámetro MODE_MODAL en la llamada a **set_mode()**).

Como último ejemplo de este apartado veremos cómo implementar la funcionalidad "Drag&Drop" en nuestros juegos. Drag&Drop (literalmente, "Arrastrar y Soltar") es el efecto de hacer clic con algún botón del ratón sobre una imagen o icono, de arrastrar esa imagen a lo largo y ancho de la pantalla (manteniendo pulsado el botón del ratón), y finalmente de soltar el botón cuando deseemos, instante en el que la imagen permanecerá fija en esa posición donde hayamos soltado el botón. "Drag&Drop" es una característica que puede dotar a nuestro programa de una gran vistosidad: es muy útil sobretodo por ejemplo para juegos de rompecabezas y puzzles.

Voy a poner ahora de golpe todo el código fuente que utilizaremos en este ejemplo, para posteriormente irlo comentando línea a línea. Si ejecutas el ejemplo verás que es sencillo: sobre un fondo aparecerán unas cuantas "fichas" que tienen cada una de ellas una letra impresa. También tendremos un icono para el ratón, el cual podremos situar encima de una ficha cualquiera y arrastrarla y soltarla allí donde nos plazca dentro de la ventana. En el código no hay implementado la opción de detectar colisiones entre fichas, pero no sería muy difícil añadirlo, y con un poco más de programación se podría conseguir un pequeño juego de puzzle. Animo fervientemente al lector a que intente llevar a cabo este proyecto tan interesante. Bien. El código fuente a estudiar es éste:

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_draw";
import "mod_grproc";
import "mod_math";

global
/*Declaramos una tabla conteniendo todos los caracteres que necesitaremos para irlos
arrastrando*/
  String example_text[8] = 'D','r','a','g','&','D','r','o','p';
  //Algunas variables de ayuda
  int graphic1;
  int graphic2;
/*Estas variables contendrán coordenadas relativas y el ID del proceso que estemos
arrastrando en ese momento.*/
  int relativeX;
  int relativeY;
  int dragID;
end

process main()
begin
/*Cambio el color de texto para que sea más bonito que no blanco*/
          set_text_color(rgb(150,150,150));
/*El fondo siempre puede ser accesible como el gráfico número 0 de la librería del
sistema (el FPG número 0)*/
          map_clear(0,0,rgb(10,10,200));
          write_var(0,0,0,0,dragID);

/*A continuación vienen las rutinas de generación dinámica de los gráficos y procesos que
los contienen. No es lo más importante del programa; se usa este sistema principalmente
porque así es más fácil enviar el código a otra persona: sólo hay que enviarle el .prg
sin necesidad de utilizar otros molestos archivos como los FPG. El código de generación
puede parecer bastante complicado a primera vista y realmente no incumbe a aquellos que
quieran aprender exclusivamente cómo funciona las bases del Drag&Drop, así que no estará
muy documentado.La idea principal para crear el puntero del ratón dinámicamente con forma
de flecha es definir unos criterios para que los puntos pertenezcan al gráfico flecha, y
si no los cumplen, éstos serán de color negro. Chequea este funcionamiento si quieres,
alterando el código como consideres oportuno para ver los posibles cambios en el
dibujo.*/

          //Dibujaremos una flecha que será el puntero del ratón:
          mouse.graph = new_map(30,30,32);
          set_center(0,mouse.graph,0,0);
          for(x=0;x<30;x++)
                    for(y=0;y<30;y++)
                              if(   (x+y   =<   30   and   abs(fget_angle(0,0,x,y)-
fget_angle(0,0,30,30)) < 25000 ) or (abs(x-y) < 4 and x+y > 20))
                                        map_put_pixel(0,mouse.graph,x,y,rgb(254,254,254));
                                    end
                    end
          end

/*Ya que 30x30 es bastante grande para un puntero de ratón en esta resolución (320x240),
muéstralo al 50% de su tamaño*/
```

```
          mouse.size = 50;

/*El código siguiente crea un gráfico de 1x1 píxel para servir como puntero efectivo del
ratón. En casos donde utilices un puntero de ratón y planees utilizar comandos de
colisión con él, a menudo es mejor utilizar dos gráficos: uno para el puntero del ratón
(típicamente una flecha) y bajo éste, un gráfico de 1x1 para chequear las colisiones.
Haciendo esto evitarás errores como que puedas arrastrar cosas con la cola de la flecha,
por ejemplo: el puntero efectivo del ratón será una porción mucho más pequeña y precisa
de lo que en realidad de ve por pantalla. En este ejemplo uso una flecha como gráfico del
puntero del ratón (en la estructura mouse) pero el programa tiene un proceso representado
por pequeño gráfico que será el puntero efectivo. Dicho gráfico está coloreado blanco
porque píxeles transparentes recuerda que no colisionan*/
          puntero();

/*Aquí es donde los gráficos con las letras se generan y los procesos "letter()" son
creados también. Uso un FOR para generar  9 procesos "letter()" -x va de 0 a 8-. Como
texto uso la tabla declarada al principio en la sección global y rellenada con letras.*/
          for(x=0;x<9;x++)
              //Creo un nuevo gráfico
              graphic1 = new_map(25,25,32);
              /*Lo pinto de gris oscuro (no negro,porque sería transparente)*/
              map_clear(0,graphic1,rgb(20,20,20,));
              /*Creo otro gráfico, negro con la letra actual (posicionada dentro de la
tabla como elemento x)*/
             graphic2 = write_in_map(0,example_text[x],4);
             /*Superpongo el gráfico negro sobre el gris oscuro, obteniendo así un dibujo
que será un cuadrado gris con una letra encima*/
              map_put(0,graphic1,graphic2,12,12);
/*Creo un nuevo proceso con este gráfico, en una posición calculada de la pantalla*/
              letter(160+(x-4)*30,120,graphic1);
          end

          loop
               if(key(_esc))exit();end
              frame;
          end
end

/* Proceso que contiene un gráfico específico en una posición específica. No mucho que
comentar*/
process letter(x,y,graph)
begin
  loop
    frame;
  end
end

/*Proceso REALMENTE importante para entender el funcionamiento del Drag&Drop.*/
process puntero()
begin
          graph = new_map(1,1,32);
          map_clear(0,graph,rgb(254,254,254));

/*Copio las coordenadas del ratón (ya que allí es donde queremos que esté el puntero
"efectivo")*/
          loop
              x = mouse.x;
              y = mouse.y;
              //Gestiona la colisión de la letra (o no si no hay ninguna)
              dragID = collision(type letter);
          /*Pero si se chequea y sí existe alguna colisión, guardo las coordenadas de la
posición relativa entre el centro de la ficha y el centro del puntero efectivo*/
              if(exists(dragID))
                     relativeX = mouse.x - dragID.x;
                     relativeY = mouse.y - dragID.y;
              end
/*Ahora necesitamos arrastrarlo mientras el jugador mantenga pulsado el botón izquierdo
del ratón*/
```

```
                while(mouse.left)
 /*Y mientras eso ocurre, necesitamos no olvidar mantener sincronizadas las coordenadas
del puntero del ratón con las de nuestro proceso, el puntero "efectivo", porque si no va
a aparecer un punto blanco fijo mientras hagamos el Drag&Drop*/
                        x = mouse.x;
                        y = mouse.y;
/*Aquí está el punto crucial. Si un proceso se detecta que colisiona a la vez que el
clic, necesita ser movido cada fotograma mientras el botón del ratón permanezca pulsado.
Esto quiere decir que las coordenadas del proceso deberían moverse con exactamente las
mismas cantidades de píxeles que el ratón.Éste es el punto crucial del programa.*/
                        if(exists(dragID))
                                dragID.x = mouse.x - relativeX;
                                dragID.y = mouse.y - relativeY;
                        end
                        if(key(_esc))exit();end
                        frame;
                end //While(mouse.left)
                frame;
        end //Loop
end
```

La línea *map_clear(0,0,rgb(10,10,200));* lo que hace simplemente es pintar el gráfico de fondo de pantalla de un color azul eléctrico. Si hubiéramos puesto otro gráfico en vez del (0,0), habríamos visto que ese gráfico era el que se pintaba entero de azul. Después de esta línea vemos que aparece un **write_var()** donde se imprime el valor de la variable *dragID*, la cual, como de momento no la hemos usado para nada, valdrá 0.

La línea *mouse.graph = new_map(30,30,32);* lo que hace es asignar a el campo ***GRAPH*** de la estructura ***MOUSE*** ( es decir, establecer el icono del puntero del ratón) a una imagen de 30x30 píxeles, todavía transparente.Por lo tanto, el siguiente paso es hacer que esta nueva imagen que se corresponderá al icono del ratón deje de ser transparente y pase a ser, por ejemplo, una flecha blanca. Es decir, ahora se trataría de "dibujar" por código la flecha. Y es lo que el programa hace a continuación, mediante la función **map_put_pixel()** dentro de los bucles FOR.

No obstante, antes de llegar allí nos encontramos con una línea que dice *set_center(0,mouse.graph,0,0);*. La línea *set_center(0,mouse.graph,0,0);* lo que hace es establecer el centro del gráfico perteneciente al puntero del ratón –el cual como lo hemos creado con **new_map()**, pertenece a la librería del sistema- en su coordenada (0,0), es decir, su esquina superior izquierda.Esto simplemente se hace para lograr tener un origen "de coordenadas" bueno para empezar los bucles FOR siguientes de forma que, tal como están diseñados, funcionen bien.La función **set_center()** la comentaremos más adelante.

Y llegamos a los bucles FOR que contienen la función **map_put_pixel()**. Esta función lo que nos hará, tal como está puesta en el código, es dibujarnos la imagen que se visualizará en la imagen –transparente en principio-, generada por **new_map()**. Fíjate dónde está **map_put_pixel()**. Dentro de dos bucles FOR y un IF. No nos pararemos a ver qué es lo que hacen exactamente todos estos bloques porque nos apartaríamos bastante del objetivo principal: lo básico que tienes que saber es que, tal como están implementados, estos bloques logran pintar píxeles blancos de tal manera que hagan una figura de flecha apuntando hacia arriba a la izquierda (donde está el "punto caliente"). Es decir, lo que hacen estos FOR e IF es realizar cálculos más o menos complejos e ingeniosos para determinar si cada uno de los píxeles que forman parte del gráfico 30x30 que representará al puntero del ratón será blanco o será transparente. Fíjate que los dos FOR no son más que dos bucles anidados para ir coordenada por coordenada, por todas las Y de cada una de las X, y en cada coordenada se comprueba la condición que determinará qué pintar. La gracia está en que al final resulta el dibujo de la flecha. La condición del If es bastante ingeniosa: puedes cambiar algún valor de éste, a ver qué ocurre.Ya tenemos el puntero del ratón dibujado, el cual se moverá allí donde lo mandemos. Pero todavía no hace nada más. De momento, fíjate que la línea siguiente lo que hace es disminuir a la mitad el tamaño visible del puntero del ratón con su propiedad *size*, para que sea un tamaño más normal:15x15.

Y ahora llegamos a un punto crucial: la creación del proceso "puntero()". Este proceso va a representar el "punto caliente" del ratón: es decir, la zona del puntero del ratón que será la que detecte colisiones con las diferentes cartas. Esta zona haremos que sea pequeña, y situada en la punta de la flecha, para que las colisiones solamente se detecten en ese punto y no en todo el gráfico de la flecha, cosa que quedaría feo. Posiblemente te estés preguntando si es necesario crear un proceso como éste. Tenemos una estructura *mouse* donde hemos definido el cursor, el tamaño, la posición dentro de la ventana,etc. ¿Por qué no podemos utilizar esta estructura para detectar colisiones? Pues precisamente, porque *mouse* no es ningún proceso, aunque se parezca. Es decir, en anteriores ejemplos hemos visto códigos que podríamos aplicar aquí, del estilo:

```
Process letter()
Private
          Int id1;
End
          …
          If(Id1=collision(type mouse))
                    …
          end
…
end
```

donde, efectivamente, el ratón detecta colisiones. Pero esto no es lo que nos interesa ahora. En el código anterior, *id1* siempre valdrá el identificador del proceso que esté entre paréntesis tras el *type.* Y este "proceso" es el ratón, así que si el puntero –o más concretamente,su punto central- choca contra el proceso "letter()" determinado cualquiera,lo único que obtendremos es que *id1* pasa a valer siempre 1, porque sólo hay un puntero. Nosotros lo que queremos es al revés: obtener el identificador concreto de la carta con la que choca el puntero. ¿Para qué? Para arrastrar sólo y exclusivamente esa carta y no las demás cuando haya Drag&Drop. Es decir, nosotros necesitaríamos algo parecido a esto:

```
Process mouse()
Private
          Int id1;
End
          …
          If(Id1=collision(type letter))
                    …
          end
…
end
```

Pero esto no puede ser porque el proceso *mouse* NO EXISTE: es una estructura. Por lo tanto, nos lo tenemos que inventar. Es decir, tendremos que crear un proceso, que asuma el papel del puntero.Este nuevo proceso tendrá asociado un gráfico, tal como he dicho, que será un cuadrado de 1x1 píxel situado en la punta de la flecha, que será precisamente la única zona que podrá detectar las colisiones, así que matamos dos pájaros de un tiro: hacemos que el puntero del ratón detecte colisiones y además que no las detecte en todo su gráfico sino sólo en su punta.Este proceso "puntero()" es el que realmente tiene todo el peso del programa.Lo comentaremos al final, una vez acabemos con el programa principal.

El FOR que viene a continuación, en el programa principal, es sencillo de entender: simplemente realiza 9 iteraciones, donde en cada una de ellas crea un gráfico 25x25 en memoria de color oscuro, que representará una ficha .Seguidamente, con **write_in_map()** se logra convertir -en cada iteración- cada una de las letras de la tabla en un gráfico más. Luego aparece la función **map_put()** para fusionar el dibujo de la carta con su letra correspondiente en un sólo dibujo.Finalmente, la última línea interna de este FOR es una llamada –repetida 9 veces- al proceso "letter()", por lo que aparecerán 9 fichas con su letra correspondiente situadas una al lado de la otra, (ya que su variable ***X*** local cambia en cada llamada).

Fíjate que el proceso "letter()" no tiene nada, y ya nuestro proceso principal tampoco. Ahora falta ver el código de nuestro proceso "puntero()", que es que realmente va a efectuar el Drag&Drop. El código de este proceso empieza creando el gráfico del proceso, un cuadrado 1x1 blanco, para meterse en un bucle infinito. En este bucle lo primero que se hace es obligar a que este cuadrado pequeño viaje siempre con el ratón, de manera que pase desapercibido y logre realizar su cometido de detector de colisiones. Y eso lo hará siempre hasta que se aprete el botón izquierdo del ratón. En ese momento,es cuando se detectará o no si hay colisión. Esta línea almacenará en *dragID* un 0 si no hay colisión (se ha pulsado el ratón en una zona sin fichas) o bien, si la hubiera, el identificador concreto de la ficha con la que se está colisionando. Seguidamente comprobamos si efectivamente *dragID* tiene el valor de algún identificador de un proceso existente (con la función **exists()**: también habríamos podido poner como hacíamos hasta ahora la línea del **collision ()** dentro del if y no usar, pues, esta función). Si no hay colisión, realmente el código no hace nada: el proceso continúa siguiendo al puntero y ya está. Pero si hay colisión, lo primero que se hace es fijar unas cantidades (*relativeX* y *relativeY*) que representarán la distancia entre el puntero del ratón y el centro de la ficha colisionada, en el momento de apretar el botón izquierdo. Este valor constante simplemente sirve para que cuando arrastremos alguna ficha, el centro de su gráfico no se acople con la punta de la flecha –es el comportamiento normal, pero queda feo- sino que la posición relativa de su centro respecto la punta de la flecha se mantenga igual al principio

del Drag&Drop correspondiente.A continuación entramos en el while que ejecuta de hecho el Drag&Drop: mientras se mantenga pulsado el botón izquierdo, las coordenadas del centro de la ficha colisionada –identificada por *dragID*- serán las mismas que las de la punta de la flecha, pero corregidas por la distancia relativa que acabamos de comentar.Y ya está. Basta procurar escribir la orden Frame en aquellos sitios que puedan hacer colgar el programa (bucles), y ya lo tenemos.

## *Trabajar con el teclado (aceptar cadenas de caracteres)

Supongo que a estas alturas te habrás preguntado si existe algún comando de Bennu que permita la introducción de cadenas de caracteres desde el teclado por parte del jugador. Un ejemplo típico sería cuando en un juego se pregunta el nombre al jugador para inscribirlo en la lista de récords de puntos. O para darle un nombre al protagonista de un RPG, o incluso, en ese RPG, para poder escribir la respuesta a alguna pregunta que se le haga al personaje. Pues bien. No existe ningún comando que haga eso. Y para poder conseguir una funcionalidad parecida, tendremos que recurrir a ciertas artimañas.

Lo que sí tiene Bennu es una variable global entera llamada ***ASCII*** que guarda en cada momento un número. Y ese número es el valor ASCII de la tecla pulsada en el último frame o un 0 si no se pulsó ninguna. Si se han pulsado varias teclas durante el frame las pulsaciones quedan almacenadas en un pequeño buffer (almacén temporal de memoria) que evita la pérdida de pulsaciones. ¿Pero qué es eso de los valores ASCII?¿Qué tiene que ver con lo que se escribe con el teclado?

Los ordenadores sólo saben trabajar con números (el 0 y el 1, y a partir de aquí, los demás), pero no con texto. Para conseguir que utilicen caracteres, todos los PC incorporan grabada a nivel de circuitería hardware una tabla, la tabla ASCII. Por ser PC, ya tienen esa tabla incrustada en los chips. Y esa tabla simplemente es una correspondencia entre los caracteres más usuales (letras minúsculas, letras mayúsculas, dígitos, signos de puntuación, espacios en blanco,etc) y un número, el llamado código ASCII para un carácter determinado. De esta manera, los ordenadores pueden trabajar con las letras como si fueran números. La tabla tiene 256 códigos, desde el 0 hasta el 255, por lo que no caben muchos caracteres, sólo los más usuales en el idioma inglés. Por ejemplo, el código ASCII del carácter 'A' (letra a mayúscula) sería el número 65, del código de 'B' sería el 66, el del dígito '0' el 48, el del dígito '1' el 49, el del carácter 'a' (letra a minúscula) el 97, el de la 'b' el 98, el del espacio en blanco el 32,etc. Si quieres saber cuál es el código ASCII de cualquier carácter es muy fácil: o haces un programa que te saque toda la lista, o bien visitas http://www.asciitable.com, o bien en cualquier editor de textos o barra de direcciones del explorador tecleas el código que quieras con el teclado numérico del teclado mientras mantienes pulsada la tecla ALT. Y ya verás que se escribe el carácter correspondiente. Por ejemplo, algunas direcciones de páginas web tienen el carácter ~. Este carácter no aparece en ninguna tecla de algunos teclados; por lo que el truco está en imprimirlo utilizando su código ASCII, que es el 126. Hay que decir, no obstante, que el estándar ASCII hace ya muchos años que ha sido mejorado y ampliado por otro estándar (bastante más complejo y que podríamos considerar que lo "incluye"): el estándar Unicode, el cual permite la representación de todos los caracteres de todos los lenguajes humanos (árabe, hebreo, chino, japonés, cirílico, etc,etc). Pero para lo que haremos nosotros, la tabla ASCII ya nos es suficiente.

Aquí tienes un posible código para obtener la lista de carácteres ASCII.

```
import "mod_map";
import "mod_key";
import "mod_video";
import "mod_string";
import "mod_text";

PROCESS MAIN()
PRIVATE
INT cx = 0;
INT cy = 0;
INT g_char = 0;
END
BEGIN
set_mode(400,400,32);
graph = new_map( 320, 320, 32);
x=200; y=200;
FROM cy = 0 TO 15;
            FROM cx = 0 TO 15;
```

```
                    g_char = write_in_map( 0, chr( (cy*16)+cx ), 0);
                    map_put( 0, graph, g_char, cx*20, cy*20);
                    unload_map( 0, g_char);
            END
END
WHILE (NOT key(_esc)) FRAME; END
END
```

Como veníamos diciendo, Bennu incorpora una variable global entera, la variable ***ASCII***, que almacena el código ASCII del carácter pulsado en el teclado en el frame actual, y si en este último frame no se ha pulsado ninguna tecla, devuelve 0. Por tanto, la variable ASCII se actualiza en cada Frame a 0 si no se ha pulsado nada o a la tecla que se haya pulsado en ese frame concreto. También hemos dicho que si se han pulsado varias teclas durante el frame, (que es lo normal porque dependiendo de la velocidad que hayamos configurado con **set_fps()** tendremos tiempo de sobra para pulsar más teclas), las pulsaciones quedan almacenadas en un pequeño buffer que evita la pérdida de pulsaciones. Por lo tanto, si se quiere recoger la entrada por teclado de un conjunto de caracteres, el truco más evidente será hacer la lectura de los caracteres de uno en uno en un bucle, donde en cada iteración el valor de la variable ***ASCII*** se imprimiría por pantalla. Y el bucle finalizaría cuando se pulsara una tecla concreta, como puede ser la tecla ENTER. Pero ojo, no confundir la tecla ENTER con el carácter ENTER (que también existe): el carácter ENTER es un carácter más como otro cualquiera de la tabla ASCII que se "imprime" cuando se pulsa la tecla ENTER, pero no es lo mismo una tecla que un carácter (en la tecla "A" se pueden imprimir dos caracteres diferentes: la a mayúscula y la a minúscula, por ejemplo).El carácter ENTER viene representado por el número 13 (retorno de carro) en la tabla ASCII.

Como muestra de lo explicado, podríamos utilizar el siguiente código:

```
import "mod_key";
import "mod_string";
import "mod_text";

PROCESS MAIN()
private
          string cadena;
end
begin
          repeat
                    if(ascii!=0)
                              cadena=cadena+chr(ascii);
                              write_var(0,100,100,4,cadena);
                    end
                    frame;
          until(key(_enter))
          repeat
                    frame;
          until(key(_esc))
end
```

Este programa hace lo siguiente: consiste en un bucle REPEAT/UNTIL que va a acabar cuando se pulse la tecla ENTER. Mientras no ocurra esto, se van a ir recogiendo los caracteres tecleados y se irán imprimiendo a la vez en tiempo real por pantalla. Para lograr esto, la línea fundamental es *cadena=cadena+chr(ascii)*. Lo que hacemos aquí es actualizar el valor de *cadena* (la variable donde almacenamos lo que tecleamos y que mostraremos por pantalla) a partir del valor que tenía antes concatenándole (acordarse de que para concatenar –"juntar" cadenas se utiliza el símbolo +) el nuevo carácter pulsado en el frame actual. Este último carácter pulsado viene dado por la expresión *chr(ascii)*. Esta expresión no es más que la función **chr()** de Bennu (definida en el módulo "mod_string"), la cual lo único que hace es convertir el código numérico ASCII que se le ponga como parámetro (en este caso el valor de la variable ***ASCII***) en su carácter correspondiente, que es lo que queremos concatenar a la cadena. Por ejemplo, chr(97) devolvería "a". Y seguidamente, una vez actualizada la cadena, se imprime por pantalla. Fíjate en el detalle de utilizar **write_var()**; se podría haber utilizado el binomio **delete_text()/write()**, pero con **write_var()** es mucho más elegante.

No dejar de mencionar la existencia de la línea frame;, imprescindible entre otras cosas para que la variable ***ASCII*** pueda actualizarse. Y por último, comentar el por qué del if; si no lo pusiéramos, el programa funcionaria igualmente, pero lo que haría es que en cada frame que no se pulsara tecla alguna, concatenaría a la cadena el carácter que tiene el código ASCII 0 –es un carácter especial que no se imprime-, porque la variable ASCII en esos frames siempre vale 0. Por tanto, estaríamos alargando inecesariamente *cadena* con caracteres inútiles hasta el infinito, y a la larga el programa petaría porque no podría manejar una cadena tan larga. Por tanto, el if sirve para detectar que si

no hay tecla pulsada en ese frame, no hace falta añadir ningún carácter a la cadena.

*Por cierto, comentamos como inciso que existe una función en Bennu, la función **asc()**, definida en el módulo "mod_string", que hace exactamente lo contrario que **chr()**. Recibe un parámetro que ha de ser un carácter, y lo que devuelve es el código numérico ASCII que le corresponde. Así, asc("a") devolvería 97.*

Volvamos donde estábamos. Verás que si ejecutas el programa anterior, desgraciadamente, tiene varios problemas. Primero, que cuando se pulsa ENTER aparece un símbolo raro. Segundo, si pulsamos la tecla de retroceso no hace retroceso ninguno y además aparece otro símbolo raro. Y tercero y más importante, que cuando pulsamos una tecla, aparecen tres o cuatro veces repetido el mismo carácter. Esto último es porque mantenemos sin querer la tecla pulsada más de un frame, y por tanto, en cada frame se realiza más de una iteración del bucle. Entonces, ¿qué podemos hacer? Mejoraremos el código de la siguiente manera:

```
import "mod_key";
import "mod_string";
import "mod_text";

process main()
private
    int letra_ant;
    string cadena;
end
begin
     loop
          if (ascii!=letra_ant)
                    switch (ascii)
                        //para que no haga nada si no se pulsa una letra
                        case 0: end
                        //para que no se imprima el carácter ENTER
                        case 13: break; end
                        //para que la tecla de retroceso haga su tarea
                        case 8 : cadena=substr(cadena,0,len(cadena)-1); end
//aquí podríamos escribir el código adecuado para el carácter TAB (tabulador)
                        case 9: end
//aquí podríamos escribir el código adecuado al carácter ESC (el de la tecla ESC)
                        case 27: end
//aquí, el código adecuado al carácter "inicio de linea" (el de la tecla Inicio)
                        case 18: end
//aquí, el código adecuado al carácter "final de linea" (el de la tecla Fin)
                        case 19: end
/*Si no se ha pulsado ninguno de los caracteres especiales anteriores, es un carácter
"normal" y por lo tanto, se imprime por pantalla*/
                        default:
                               cadena=cadena+chr(ascii);
                               write_var(0,100,100,4,cadena);
                        end
                    end
          end
          letra_ant=ascii;
          frame;
    end
end
```

Este código está bastante mejor.Lo que se hace es tener un bucle infinito del cual sólo podremos salir si apretamos el carácter 13 (ENTER). Si no apretamos nada en un frame, fíjate que el case correspondiente (Case 0) no hace nada de nada. También he incluido el esqueleto para dar la posibilidad más adelante de que el programa haga diferentes cosas para los caracteres tabulador,ESC,inicio y fin de línea, ya que si te fijas, éstos continuan mostrando signos raros en pantalla cuando se pulsan y no hacen lo que presumiblemente deberían hacer (tabular y salir, por ejemplo).

Si se pulsa la tecla de retroceso, que tiene el código 8, fíjate lo que hacemos para que realice su función. Le damos un nuevo valor a la variable *cadena*, que será el que tenía menos el último carácter. Y para hacer eso utilizamos el comando **substr()**, ya visto anteriormente en alguna ocasión peregrina, pero que ahora vamos a comentar en profundidad.

*Este comando precisamente sirve para devolver un trocito de cadena (una subcadena) a partir de una cadena más grande.Tiene tres parámetros:la cadena original, un número que es la posición del primer carácter que se quiere "extraer" de la cadena original por el cual comenzará la subcadena (empezando siempre por el 0), y otro número que es el número de caracteres que se quiere que tenga la subcadena, cogiendo los caracteres a partir del primero extraido.*

*Es más fácil verlo con un ejemplo. Si tenemos la línea substring("Hola qué tal",2,3), lo que devuelve esta línea será la cadena "la ". Porque lo que se ha hecho es, a partir de la cadena original, ir al carácter que ocupa la posición 2, y a partir de él, coger los tres caracteres siguientes. Como el primer carácter es el número 0, el carácter 2 es la ele. Y como hemos dicho que se quieren coger 3 caracteres, se coge la ele, la a y el espacio en blanco. Y ésta es la subcadena obtenida. Por lo tanto, ¿qué hace la línea cadena=substr(cadena,0,len(cadena)-1);? Pues que a partir del primer carácter de cadena (el texto que se ha ido imprimiendo) se obtiene una subcadena con un determinado número de caracteres.¿Cuántos? Lo que nos dé la expresión len(cadena)-1. Recuerda que* ***len()*** *lo que hace es devolver el número de caracteres que tiene una cadena pasada como parámetro. Así que ya lo tenemos. Si ponemos como tercer parámetro de* ***substr()*** *esto: len(cadena)-1, lo que estamos diciendo es que "extraeremos" para la subcadena que queremos crear un número de caracteres tal que sea el mismo que el de la cadena original menos uno.Si empezamos a "extraerlos" desde el principio (segundo parámetro igual a 0), lo que tendremos es pues, una subcadena igual a la cadena original pero con un carácter menos (el del final).*

*Los parámetros de* ***substr()*** *pueden tener distintos valores que tienen distintos significados concretos. Por ejemplo, si el segundo parámetro (el que indica el carácter de inicio de la subcadena) es un número negativo, estaremos especificando la posición de ese caracter de inicio pero desde el final: -1 se refiere al último caracter de la cadena, -2 al anterior, y así sucesivamente. Respecto el tercer parámetro (el que indica el número de caracteres que compondrán la subcadena), se puede omitir, en cuyo caso* ***substr()*** *tomará todos los caracteres que haya desde la posición especificada en el segundo parámetro hasta el final de la cadena siempre. Y si este tercer parámetro es un número negativo,quiere decir lo siguiente: -1 significa el último carácter (sería lo equivalente a len(cadena)), -2 el penúltimo carácter, -3 el antepenúltimo carácter, y así. Por tanto, en vez de len(cadena)-1 podríamos haber puesto -2 con el mismo resultado.*

*Un ejemplo práctico donde puedes observar los efectos de este comando, si lo ejecutas, sería éste:*

```
import "mod_key";
import "mod_string";
import "mod_text";

process main()
private
  string Txt="Hola Mundo!";
end
begin
  write(0,10,30,3,"Texto = Hola Mundo!");
  write(0,10,40,3,"SUBSTR(  0,  3) = "+substr(Txt,0,3));
  write(0,10,50,3,"SUBSTR(  3,  0) = "+substr(Txt,3,0));
  write(0,10,60,3,"SUBSTR( -1,  5) = "+substr(Txt,-1,5));
  write(0,10,70,3,"SUBSTR(  5, -1) = "+substr(Txt,5,-1));
  write(0,10,80,3,"SUBSTR( -3, -1) = "+substr(Txt,-3,-1));
  write(0,10,90,3,"SUBSTR( -1, -3) = "+substr(Txt,-1,-3));
  write(0,10,100,3,"SUBSTR(  9, -1) = "+substr(Txt,9,-1));
  write(0,10,110,3,"SUBSTR( -1,  9) = "+substr(Txt,-1,9));
  write(0,10,120,3,"SUBSTR( -9, -1) = "+substr(Txt,-9,-1));
  write(0,10,130,3,"SUBSTR( -1, -9) = "+substr(Txt,-1,-9));
  repeat
    frame;
  until(key(_esc))
end
```

Volviendo al código, ves que si no se pulsa ninguna de estos caracteres especiales, es entonces cuando ejecutamos el default, y por tanto, imprimimos por pantalla el carácter adecuado.

Por último, he de comentar el *if(ascii!=letra_ant)*. Este if es el responsable de que no aparezcan los

caracteres repetidas veces cuando pulsamos una tecla. Fíjate que en cada fotograma (antes de la línea frame;) está la línea *letra_ant=ascii;* Con esta línea lo que hacemos es utilizar una variable de refuerzo para almacenar el código ASCII del carácter pulsado en el frame actual –justo antes de pasar al siguiente-. De tal manera, que cuando se vuelve a comprobar la condición del Switch, como estamos ya en el siguiente fotograma (la línea frame; se ha acabado de ejecutar), la variable ASCII ya valdrá otro valor. Y lo que comprueba el if es si este nuevo valor de ASCII coincide con el valor del frame anterior, guardado en *letra_ant*. Si es igual, no hace nada de nada y se va directamente a la siguiente iteración del Loop, y así hasta que por fin se detecte que el valor de la variable ***ASCII*** ha cambiado (porque ya no se pulsa ninguna tecla o porque se pulsa otra). El pequeño inconveniente de este código es que no detectará palabras que tengan dos caracteres iguales seguidos ("lluvia" la tomará por "luvia", "llaves" como "laves", etc).

Otro detalle (que es fácil modificar: lo dejo como ejercicio) es evitar dejar que el usuario pueda escribir cadenas infinitas de caracteres, como hasta ahora permite el programa. Simplemente habría que comprobar que la longitud de la cadena introducida hasta el momento fuera menor que un límite máximo fijado por nosotros.

Finalmente, sería muy interesante hacer de este código, (con las modificaciones que tú creas oportunas) una función, con los parámetros que tú consideres, de manera que cada vez que se necesite pedir al jugador que escriba algún texto no se tenga que escribir siempre la parrafada anterior sino que pudieras invocar en una sola línea a una función que contuviera este código listo para ejecutar. .Esto último es lo que se ha pretendido con el código siguiente. La finalidad es la misma que el anterior, recoger la entrada por teclado, pero este ejemplo es más sofisticado, ya que también detecta el tiempo de pulsación de las teclas para cerciorarse de que es una pulsación "válida", no accidental (por ejemplo, para detectar que efectivamente queremos pulsar la tecla varias veces seguidas).

```
import "mod_key";
import "mod_string";
import "mod_text";
import "mod_video";
import "mod_say";

Declare String textinput()
End

process main()
Begin
    set_fps(60,0);
    say("Escribiste: '" + textinput() + "'");
    Repeat
        frame;
    Until(key(_esc));
End

//Necesita tener importados los módulos mod_string y mod_text
Function String textinput()
Private
    String str;
    int t;
    int t2;
    byte last_ascii;
    int txtid;
Begin
    //Limpia la cadena
    str = "";
    //Muestra lo que escribes en la esquina superior izquierda
    txtid = write_var(0,0,0,0,str);
    //Recoge la entrada del usuario. Pulsando Enter el loop termina.
    Loop
//Chequea si una tecla es presionada y si la misma tecla fue presionada en el último frame
        if(ascii!=0&&last_ascii==ascii)
//Chequea si la tecla fue presionada levemente o si ha sido presionada más de 0.25 segundos
            if(t==0||t>fps/4)
//Chequea si la tecla fue presionada levemente o si ha sido presionada en los últimos 0.03 segundos
                if(t==0||t2>fps/30)
                    t2=0;
                    switch(ascii) // Gestionar entrada
                        case 8: //Tecla retroceso (backspace)
                            str = substr(str,0,len(str)-1);
                        end
```

```
                        case 13: //Tecla enter
                            break;
                        end
                        default: //Añade la tecla
                            str=str+chr(ascii);
                        end
                    end
                end
                t2++;
            end
            t++;
        else
            t = t2 = 0; // Reset
        end
        last_ascii = ascii;
        frame;
    End
    //Borrar el texto utilizado
    delete_text(txtid);
    //Retorna la cadena tecleada
    return str;
End
```

Fíjate que hemos tenido que declarar la función *textinput()* antes del proceso "Main()", para que la función pueda retornar correctamente un tipo de datos String, tal como se comentó en el capítulo 4.

Por otro lado, finalmente queremos resaltar la diferencia que existe entre la variable global predefinida ***ASCII***, que ya conocemos, y otra variable global predefinida que conviene no confundir, ***SCAN_CODE***. El valor de ésta última será siempre un código numérico que representa la última tecla del teclado que se ha pulsado. Cada tecla del teclado es reconocida por el sistema (y por Bennu) gracias a que cuando es pulsada emite una determinada señal eléctrica que es detectada e identificada, pero es importante recalcar que estamos hablando de teclas. Por lo tanto, ***SCAN_CODE*** valdrá exactamente lo mismo tanto si escribimos la "a" minúscula como si escribimos la "A" mayúscula, porque estaremos pulsando la misma tecla (no le ocurrirá lo mismo en cambio a la variable ***ASCII***, que en los dos casos valdrá un valor diferente).

De hecho, los valores numéricos que equivalen a las constantes que siempre hacemos servir como parámetro de la función **key()** (los cuales se listaron en una tabla en el capítulo 3) son precisamente el valor de ***SCAN_CODE***. Así pues, podemos concluir que la función **key()** no hace más que detectar si el valor de ***SCAN_CODE*** es uno determinado. Por ejemplo, una línea como *if(key(_esc))*, la cual equivale a *if(key(1))* (porque _esc equivale a 1, consulta la tabla referida ) sería lo mismo que escribir *if(scan_code == 1)*

A continuación, se presenta un ejemplo donde se puede ver la diferencia entre ambas variables:

```
import "mod_text";
import "mod_key";

process main()
private
  int keys[128];
  int i;
end
begin
  write(0,0,30,0,"Codigos de teclas pulsados:");
/*Pinto una rejilla de valores, inicialmente a 0, que mostrarán el código de las teclas
pulsadas(scan_code).El sistema de colocación en pantalla de los valores es lo de menos.*/
  from i=0 to 127;
    write_var(0,(i%16)*20,(i/16)*10+40,0,keys[i]);
  end
  write(0,0,150,0,"Codigo de ultimo tecla pulsado (scan_code):");
  write_var(0,320,150,2,scan_code);
  write(0,0,160,0,"Codigo ASCII de ultimo tecla (ascii):");
  write_var(0,320,160,2,ascii);
  write(0,0,170,0,"Estado de los teclas especiales (shift_status):");
  write_var(0,320,170,2,shift_status);
  repeat
```

```
/*Detecto si se pulsa alguna tecla, en cuyo caso muestro su scan_code en la rejilla, y si
no, reseteo su lugar en la rejilla a 0. Como valores de la función key, en vez de
utilizar las constantes que solemos usar, utilizaremos su valor numérico real, para poder
implementar un bucle con él. Si no, deberíamos de escribir múltiples veces las mismas
líneas, para cada una de las teclas.*/
    from i=0 to 127;
       if(key(i)) keys[i]=i; else keys[i]=0; end
    end
    frame;
  until(key(_esc) && key(_F1))
end
```

Como se puede comprobar, cuando se pulsa una tecla cualquiera, en la rejilla aparecerá su correspondiente valor de ***SCAN_CODE***, además de mostrarse también en una línea de abajo. También se muestra el valor correspondiente de ***ASCII***. Si por ejemplo, escribes una "a" minúscula, verás que su valor de ***SCAN_CODE*** es 30 y el de ***ASCII*** es 97, pero si escribes una "A" mayúscula verás que el valor de ***SCAN_CODE*** no varía porque la tecla es la misma (aunque verás que en la rejilla también aparece el valor 54, correspondiente a la tecla MAYUS, que también la tenemos pulsada) pero el valor de ***ASCII*** sí, pasa a ser 65 debido a que el caracter es diferente.

En este ejemplo también podrás observar el uso de otra variable global predefinida: ***SHIFT_STATUS***. Esta variable indica si se mantienen pulsadas algunas de las teclas "de control" del teclado: ALT, CTRL, MAYUS, ALTGR, etc. Si no hay pulsada ninguna de estas teclas, su valor será 0. En la tabla siguiente se lista los posibles valores que podría tener dependiendo de la tecla de control pulsada. Si se mantienen dos o más teclas de éstas pulsadas a la vez, el valor de ***SHIFT_STATUS*** es la suma de los valores correspondientes. Por ejemplo: si se tiene pulsada la tecla CTRL y a la vez la de ALT, ***SHIFT_STATUS*** valdrá (tal como se puede deducir de la tabla), 12.

| | |
|---|---|
| **MAYUS derecha** | 1 |
| **MAYUS izquierda** | 2 |
| **CTRL (cualquiera)** | 4 |
| **ALT (cualquiera)** | 8 |
| **ALT GR** | 12 |
| **CTRL derecho** | 16 |
| **CTRL izquierdo** | 32 |
| **ALT derecho** | 64 |
| **ALT izquierdo** | 128 |
| **NUM LOCK** | 256 |
| **CAPS LOCK** | 512 |
| **MAYUS (cualquiera)** | 1024 |

## *Trabajar con ángulos y distancias

Una de los conceptos más elementales que hay que dominar para lograr realizar videojuegos que sean mínimamente atractivos es el dominio de la medida de los ángulos y de las distancias existentes entre los diferentes procesos que en un momento determinado se visualicen en pantalla. Echando mano de las funciones que Bennu dispone en este campo, lograremos efectos tan básicos como conseguir que un proceso esté encarado siempre hacia otro, o que un proceso persiga constantemente a otro proceso, etc,etc

| **FGET_ANGLE( X1, Y1, X2, Y2 )** |
|---|
| Esta función, definida en el módulo "mod_math", devuelve, en milésimas de grado, el ángulo formado por la línea entre los dos puntos especificados, y el eje X (horizontal) de coordenadas. |
| Con esta función se puede, fácilmente, orientar el ángulo de un proceso para que éste siempre esté "mirando" hacia un |

lugar concreto (como puede ser otro proceso). La aplicación más sencilla puede ser un proceso que "persiga" a otro simplemente utilizando **fget_angle()** seguida de **advance()**

PARAMETROS:
INT X1: Coordenada X del primer punto.
INT Y1: Coordenada Y del primer punto
INT X2: Coordenada X del segundo punto
INT Y2: Coordenada Y del segundo punto.

**FGET_DIST( X1, Y1, X2, Y2 )**

Esta función, definida en el módulo "mod_math", devuelve la distancia entre dos puntos, en las mismas unidades que vengan dados éstos.Es una función relativamente lenta, que devuelve un resultado exacto empleando una raíz cuadrada.

PARAMETROS:
INT X1: Coordenada X del primer punto.
INT Y1: Coordenada Y del primer punto
INT X2: Coordenada X del segundo punto
INT Y2: Coordenada Y del segundo punto.

**GET_ANGLE(ID)**

Esta función, definida en el módulo "mod_grproc", devuelve el ángulo formado por la línea que parte del centro del proceso actual y pasa por el centro del proceso cuyo identificador se especifica como parámetro.

El uso habitual de esta función consiste en hacer que un proceso "apunte" en dirección a otro: es el caso típico de monstruos y otros objetos que buscan siempre atrapar al jugador.

Para que esta función devuelva un resultado útil, hay que asegurarse de no asignar valores diferentes de la variable ***RESOLUTION*** y de que las coordenadas de ambos representen la misma zona (por ejemplo, que no estén cada uno en un área de scroll diferente).

PARAMETROS:
INT ID: Identificador de un proceso

**GET_DIST(ID)**

Esta función, definida en el módulo "mod_grproc", devuelve la distancia en línea recta entre el centro del proceso actual y el centro del proceso cuyo identificador se especifica como parámetro.

PARAMETROS:
INT ID: Identificador de un proceso

**GET_DISTX(ANGULO, DISTANCIA)**

Esta función, definida en el módulo "mod_math", devuelve, en las misma unidad que se use para la distancia, el ancho del rectángulo formado por la línea que recorre esa distancia. Usando esta función junto con **get_disty()** se puede saber qué cantidades sumar a las coordenadas X e Y de un objeto para desplazarlo en la dirección que se desee.

PARAMETROS:

INT ANGULO: Ángulo en milésimas de grado (90000= 90º)
INT DISTANCIA : Magnitud de distancia

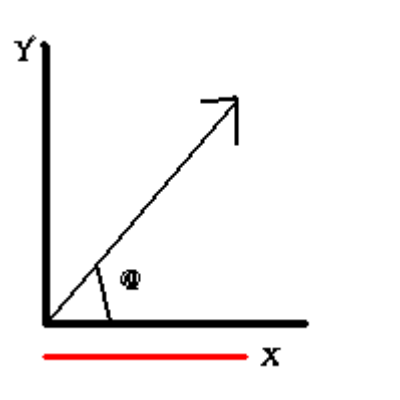



*Gráfico que ilustra, dada una distancia y un ángulo, el valor "x" que devuelve la función get_distx() (lo marcado en rojo)*

**GET_DISTY(ANGULO, DISTANCIA)**

Esta función, definida en el módulo "mod_math", devuelve, en las misma unidad que uses para la distancia, la altura del rectángulo formado por la línea que recorre esa distancia. Usando esta función junto con **get_distx()** se puede saber qué cantidades sumar a las coordenadas X e Y de un objeto para desplazarlo en la dirección que se desee.

PARAMETROS:

INT ANGULO: Ángulo en milésimas de grado (90000= 90º)
INT DISTANCIA : Magnitud de distancia

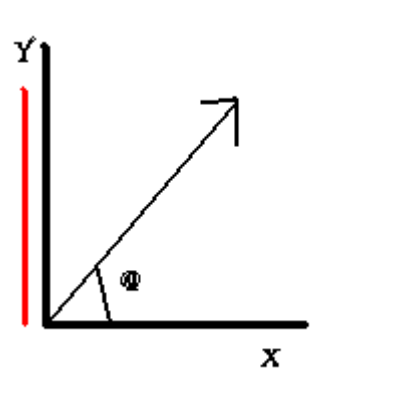



*Gráfico que ilustra, dada una distancia y un ángulo, el valor "y" que devuelve la función get_disty() (lo marcado en rojo)*

A continuación vamos a ver unos cuantos ejemplos básicos que ilustran el funcionamiento de las funciones anteriores. Por ejemplo, el siguiente código va mostrando a cada segundo una línea diferente cuyos dos extremos son puntos de coordenadas aleatorias. Para cada línea que aparezca el programa nos dirá el ángulo que forma ésta respecto el eje X (con **fget_angle()**) y su longitud -es decir, la distancia entre sus extremos- (con **fget_dist()**).

```
import "mod_key";
import "mod_proc";
import "mod_text";
import "mod_grproc";
import "mod_math";
import "mod_rand";
import "mod_draw";
import "mod_video";

process main()
private
  int xorigen,yorigen,xfinal,yfinal;
  int distancia,angulo;
end
begin
```

```
  drawing_z(0);
  set_fps(1,0);
  repeat
     delete_text(all_text);
     xorigen=rand(10,300);
     yorigen=rand(10,200);
     xfinal=rand(10,300);
     yfinal=rand(10,200);

     delete_draw(0);
     draw_line(xorigen,yorigen,xfinal,yfinal);
//Las cajitas hacen un poco más visibles los dos extremos de la línea
     draw_box(xorigen-3,yorigen-3,xorigen+3,yorigen+3);
     draw_box(xfinal-3,yfinal-3,xfinal+3,yfinal+3);

     distancia=fget_dist(xorigen,yorigen,xfinal,yfinal);
     write(0,0,220,0,"Distancia entre los puntos: " + distancia);
     angulo=fget_angle(xorigen,yorigen,xfinal,yfinal);
     write(0,0,230,0,"Angulo formado por la línea y el eje X: "+ (float)angulo/1000);
     frame;
  until(key(_esc))
end
```

Para mostrar la utilidad de las funciones **get_dist()** y **get_angle()** es interesante estudiar el ejemplo siguiente. En él aparecen dos procesos: un cuadrado verde inmóvil y un cuadrado rojo que podremos mover con los cursores. En todo momento aparecerán por pantalla dos datos, que serán los devueltos por **get_dist()** y **get_angle()**, respectivamente. Es decir, la distancia a la que se encuentra en cada momento el cuadrado rojo que dominamos respecto el cuadrado verde, y el ángulo que forma la línea que une ambos cuadrados respecto la horizontal -ambas líneas también aparecen pintadas de blanco para facilitar la interpretación de dicho número-

```
import "mod_key";
import "mod_proc";
import "mod_text";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";

global
  int idverde;
end

process main()
begin
  set_mode(640,480);
/*El identificador idverde lo necesitaremos para las funciones get_dist y get_angle,
ejecutadas desde rojo()*/
  idverde=verde();
  rojo();
  while(!key(_esc))
     frame;
  end
  let_me_alone();
end

process rojo()
private
   int angulo,distancia;
end
begin
  graph=new_map(30,30,32);
  map_clear(0,graph,rgb(255,0,0));
  x=400;
  y=100;
  drawing_z(0);
  loop
```

```
    if(key(_left)) x=x-10; end
    if(key(_right)) x=x+10; end
    if(key(_up)) y=y-10; end
    if(key(_down)) y=y+10; end

/*Dibujo la horizontal y la línea que une los dos procesos, para mostrar visualmente el
significado del valor devuelto por get_angle. Fijarse como aquí utilizo ya el
identificador idverde para acceder a las coordenadas de posición de un proceso diferente
al actual.*/
    delete_draw(0);
    draw_line(x,y,x+100,y);
    draw_line(x,y,idverde.x,idverde.y);

    delete_text(0);
    distancia=get_dist(idverde);
    write(0,200,200,4,"La distancia al cuadrado verde es de " + distancia);
    angulo=get_angle(idverde);
/*Es curioso comprobar que los valores que devuelve get_angle van de 0º a 270º, y a
partir de allí de -90º a 0º otra vez*/
    write(0,200,250,4,"El ángulo que formo respecto el cuadrado verde es " +
(float)angulo/1000);
    frame;
  end
end

process verde()
begin
  graph=new_map(30,30,32);
  map_clear(0,graph,rgb(0,255,0));
  x=500;
  y=100;
  loop
     frame;
  end
end
```

Si queremos ver la utilidad de las funciones **get_disty()** e **get_distx()**, podemos aprovechar el ejemplo anterior y simplemente añadir, justo antes de la orden frame del loop/end del proceso rojo() las siguientes líneas:

```
distx=get_distx(angulo,distancia);
disty=get_disty(angulo,distancia);
write(0,200,300,4,"La distancia horizontal al cuadrado verde es de " + distx);
write(0,200,350,4,"La distancia vertical al cuadrado verde es de " + disty);
```

teniendo la preocupación además de declarar como variables privadas del proceso rojo() las variables *distx* y *disty*.

Si pruebas ahora el código, verás que obtienes dos datos más: los dos datos devueltos por las funciones **get_distx()** y **get_disty()** respectivamente, a partir de un ángulo y una distancia dados, los cuales hemos puesto que sean para redondear el ejemplo precisamente el ángulo y la distancia existentes entre los dos procesos del ejemplo. Fíjate que, evidentemente, si movemos el cuadrado rojo en horizontal, el valor devuelto por **get_disty()** no se altera, y al revés: si movemos el cuadrado rojo en vertical, el valor devuelto por **get_distx()** tampoco cambia, como ha de ser.

Otro ejemplo ilustrativo del uso de **get_angle()**, **get_disx()** y **get_disty()** es el siguiente. En él tenemos un proceso "cuadamar()" visualizado por un cuadrado amarillo el cual podremos mover con el cursor y que es perseguido incansablemente por otro proceso, "cuadroj()" (visualizado por un cuadrado rojo). La idea es que cuando "cuadroj()"se sale de la pantalla se vuelve a colocar en una nueva posición aleatoria dentro de la pantalla para continuar la persecución. Esta persecución, no obstante, es sui generis, porque "cuadroj()" sólamente tiene en cuenta la posición que tiene "cuadamar()" en el momento de su creación -o su puesta de nuevo dentro de la pantalla-, por lo que si "cuadamar()" se mueve, "cuadroj" no lo sabrá y seguirá imperturbable en la dirección que "cuadamar()" tenía originalmente.

```
import "mod_key";
import "mod_proc";
import "mod_text";
import "mod_grproc";
```

```
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";
import "mod_rand";

Global
    int idcuadamar;
End

process main()
Begin
    set_mode(640,480);
    write(0,0,20,0,"TECLAS RIGHT,LEFT,UP y DOWN para mover cuadrado amarillo");
    idcuadamar=cuadamar();
/*El cuarto parámetro de cuadroj hace que a la hora de crear dicho proceso, se obtenga el
ángulo que en ese instante tiene cuadamar respecto el nuevo proceso cuadroj. Este ángulo
se utilizará para dirigir cuadroj a la posición que ocupa cuadamar, pero como esta
medición sólo se hace en este instante -cuando se crea cuadroj-, si posteriormente
cuadamar se mueve, cuadroj no se enterará y continuará estando dirigido en la dirección
que ocupaba cuadamar originalmente.*/
    cuadroj(x,y,get_angle(idcuadamar));
End

Process cuadroj(x,y, int angulo)
   Begin
    graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
    Loop
/*A partir del ángulo que hay entre cuadamar y cuadroj, se calcula las nuevas coordenadas
de cuadroj*/
        x=x+get_distx(angulo,16);
        y=y+get_disty(angulo,16);
/*Si cuadroj sale de los límites de la pantalla, se vuelve a colocar en unas coordenadas
aleatorias otra vez dentro, y se vuelve a obtener la nueva orientación (el nuevo ángulo)
de cuadroj respecto cuadamar (el cual también puede haber cambiado de posición mientras
tanto) como si "cuadroj" hubiera sido creado de nuevo, para volver a empezar la
persecución*/
        If(x>640 OR y>480 OR x<0 OR y<0)
                    x=rand(0,600);y=rand(0,450);
                    angulo=get_angle(idcuadamar);
        End
        If(key(_esc)) exit();End
       Frame;
   End
End

Process cuadamar()
Begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
  x=160;y=100;
  Loop
          If(key(_right)) x=x+16;End
          If(key(_left)) x=x-16;End
          If(key(_up)) y=y-16;End
          If(key(_down)) y=y+16;End
          //No se puede salir de los extremos de la pantalla
          If(x>620) x=620;End
          If(x< 20)  x=20;End
          If(y>460) y=460;End
          If(y< 20)  y=20;End
          Frame;
   End
End
```

Otro ejemplo, a lo mejor más sencillo, de estas mismas funciones es el siguiente código, donde se puede ver un cuadrado rojo que hace de "escudo vigilante" a un cuadrado amarillo.

```
import "mod_key";
```

```
import "mod_proc";
import "mod_text";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";

process main()
Private
    int distancia=100; //Distancia del cuadrado rojo al amarillo
    int idcuadamar;
end
Begin
    set_mode(640,480);
//El programa principal representa el proceso del cuadrado rojo
    graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
    idcuadamar=cuadamar();
    Loop
/*Cambiamos las coordenadas del cuadrado rojo (y su orientación). Si coges lápiz y papel
verás que es fácil deducir las fórmulas que se emplean para simular el movimiento
circulatorio*/
        angle=angle+2000;
        x=idcuadamar.x+get_distx(angle,distancia);
        y=idcuadamar.y+get_disty(angle,distancia);
        If(key(_esc)) exit();End
        Frame;
    End
End

Process cuadamar()
 Begin
  x=250;y=240;
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
  loop
//Aunque me mueva, el cuadrado rojo va a continuar rodeándome
      if(key(_left))x=x-10;end
      if(key(_right))x=x+10;end
      if(key(_up))y=y-10;end
      if(key(_down))y=y+10;end
      Frame;
 end
End
```

| **XADVANCE(ANGULO,NUMERO)** |
|---|
| Esta función, definida en el módulo "mod_grproc", desplaza el proceso actual en la dirección especificada por el ángulo indicado como primer parámetro, el número de unidades indicado como segundo parámetro.<br><br>Hay que tener en cuenta que el desplazamiento se hace en unidades enteras, y por lo tanto tiene limitaciones de precisión. Un desplazamiento de 1 ó 2 unidades no puede reflejar las diferencias entre ángulos, por lo que no será efectivo. Si se desea desplazar un proceso en ángulos arbitrarios y sólo unos pocos pixels, será preciso utilizar la variable ***RESOLUTION*** para que las coordenadas del gráfico vengan expresadas en unidades más pequeñas que un pixel, como ocurre por defecto.<br><br>PARAMETROS:<br><br>INT ANGULO: Ángulo en milésimas de grado (90000= 90º)<br>INT NUMERO : Número entero |

La función anterior se utiliza bastante para conseguir que un proceso se dirija hacia la posición que ocupa otro proceso en particular. Sabemos que **xadvance()** necesita un ángulo para dirigir el proceso actual; pues bien, en este caso, el ángulo que necesitamos sería el devuelto por la función **get_angle()**, poniendo como parámetro de dicha función el identificador del proceso "perseguido", ya que recordemos que esta función devuelve el ángulo que forma la línea que atraviesa ambos procesos respecto la horizontal: precisamente el ángulo que queremos.

Un ejemplo de esta aplicación sería el siguiente. En él tenemos dos procesos: un cuadrado rojo que persigue a otro, un cuadrado verde inmóvil que cuando es alcanzado cambia de posición aleatoriamente para continuar siendo perseguido sin descanso por el cuadrado rojo.

```
import "mod_key";
import "mod_proc";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_rand";
import "mod_video";

global
  int idverde;
end

process main()
begin
  set_mode(640,480);
  idverde=verde();
  rojo();
  repeat
     frame;
  until(key(_esc))
  exit();
end

process rojo()
private
   int angulo;
end
begin
  graph=new_map(30,30,32);
  map_clear(0,graph,rgb(255,0,0));
  x=100;
  y=400;
  loop
//Si yo, cuadrado rojo, colisiono con el cuadrado verde, hago que éste cambie de posición
    if(collision(idverde))
      idverde.x=rand(0,599);
      idverde.y=rand(0,399);
    end
//Las dos líneas importantes que realizan la persecución
    angulo=fget_angle(x,y,idverde.x,idverde.y);
    xadvance(angulo,10);
    frame;
  end
end

process verde()
begin
  graph=new_map(30,30,32);
  map_clear(0,graph,rgb(0,255,0));
  x=500;
  y=100;
  loop
     frame;
  end
end
```

Fíjate, no obstante, que a diferencia de la explicación inicial, en el código no hemos utilizado la función **get_angle()** para averiguar el ángulo que forman los dos procesos entre ellos respecto la horizontal, sino que hemos hecho servir la función **fget_angle()**, especificando "a mano" los centros de ambos procesos. Las dos son funciones diferentes con propósitos diferentes, pero en este ejemplo podríamos usar una u otra según nos interese, y haciéndolo bien obtendríamos el mismo resultado. ¿Cómo se podría modificar el código anterior para poder utilizar en vez de **fget_angle()**, el comando **get_angle()**?

Existen otras posibilidades a la hora de implementar mecanismos de persecución de procesos, a parte del más típico comentado en el apartado anterior cuando se hablaba de la función **xadvance()**.En ocasiones nos puede interesar, por ejemplo, que un proceso, a partir de colisionar con otro, vaya siguiendo los pasos de éste. ¿Cómo podríamos lograr este efecto?

A continuación se presenta un ejemplo que de hecho no hace ningún uso de las funciones acabadas de comentar, sino que utiliza métodos "más manuales", pero igualmente efectivos. La idea es simple: tenemos dos procesos: un cuadrado amarillo que será el proceso perseguido y el cual podremos mover con los cursores y un cuadrado blanco que permanecerá inmóvil hasta que el cuadrado amarillo colisione con él, momento en el cual el cuadrado blanco comenzará a perseguir al amarillo pisándole los talones para siempre. Lo primero que hemos tenido que preguntarnos es en qué proceso escribimos el comando **collision()**. ¿El cuadrado amarillo colisiona con el blanco, o viceversa? En realidad, si sólo vamos a incluir estos dos procesos, la respuesta es irrelevante porque tanto da, pero si pensamos en una posible ampliación del programa donde haya más de un proceso potencialmente a perseguir por parte del cuadrado blanco, la respuesta sólo tiene una posibilidad. Si queremos centralizar todas las colisiones que ocurran dentro de nuestro programa entre el único cuadrado blanco y los múltiples procesos a perseguir, el comando **collision()** lo deberemos de escribir en el proceso correspondiente al cuadrado blanco, porque así podremos gestionar las colisiones que sufre éste con cualquiera de los otros procesos; si hiciéramos al revés, deberíamos de poner un comando **collision()** en cada uno de los procesos perseguidos comprobando su respectiva colisión con el cuadrado blanco, cosa que no es nada óptimo. De momento, sin enbargo, sólo incluiremos en nuestro ejemplo un proceso perseguido (el cuadrado amarillo), y posteriormente veremos cómo introducir más procesos de este tipo, acción que si lo hemos hecho bien desde el principio, será trivial.Una vez resuelto dónde colocamos el comando **collision()**, fijémosnos en el código.

```
import "mod_key";
import "mod_proc";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_rand";
import "mod_video";

process main()
begin
  set_mode(640,480,32);
  perseguido1();
  perseguidor();
  repeat
    frame;
  until(key(_esc))
  exit();
end

process perseguido1()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
  x=300;y=300;
  loop
      if(key(_left)) x=x-10;end
      if(key(_right))x=x+10;end
      if(key(_up))   y=y-10;end
      if(key(_down)) y=y+10;end
      frame;
  end
end

process perseguidor()
private
   int idprota;
   int flagseguir=0;
end
begin
   graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
   x=200;y=200;
   loop
```

```
      if(idprota=collision(type perseguido1))
              flagseguir=idprota;
      end
 /*Los +20 y -20 son para que el perseguidor no se coloque encima del perseguido, sino
que se quede al lado.Estos ifs lo que hacen es comprobar si el perseguido está a la
derecha/izquierda/arriba/abajo respecto el perseguido, para moverse en consecuencia.
Fijarse que el perseguidor va un poco más lento que el  perseguido, para causar un efecto
chulo.*/
      if (flagseguir!=0)
              if (x+20<flagseguir.x)x=x+7;end
              if (x-20>flagseguir.x)x=x-7;end
              if (y+20<flagseguir.y)y=y+7;end
              if (y-20>flagseguir.y)y=y-7;end
      end
      frame;
   end
end
```

Lo único digno de mención es el proceso perseguidor. Su punto interesante está en el uso de la variable global *flagseguir*: mientras el cuadrado amarillo no colisione con el blanco, esta variable valdrá 0 y mientras valga este valor, no ocurrirá nada. En el momento de la colisión, *flagseguir* pasa a valer el identificador del proceso con el que se ha colisionado -en este caso, el identificador del cuadrado amarillo-, y es entonces cuando el código que hay en el interior del if (ya que *flagseguir* ya no vale 0) se ejecuta, posibilitando así la persecución, puesto que *flagseguir* , insisto, vale el identificador del proceso que acaba de colisionar (es decir, el proceso a perseguir).

Un detalle importante: ¿por qué NO se ha programado el loop del proceso perseguidor así?

```
loop
      if(idprota=collision(type perseguido1))flagseguir=1; end
      if (flagseguir!=0)
              if (x+20<idprota.x)x=x+7;end
              if (x-20>idprota.x)x=x-7;end
              if (y+20<idprota.y)y=y+7;end
              if (y-20>idprota.y)y=y-7;end
      end
      frame;
end
```

Si pruebas el cambio verás que el programa deja de funcionar justo en el momento que se produce una colisión. Y esto ocurre por lo siguiente: justo en ese momento activamos *flagseguir* para señalar que ha habido colisión y además el comando **collision()** devuelve el identificador del cuadrado amarillo y , tal como hemos hecho, lo guardamos en la variable *idprota*. Pero esto último sólo ocurre mientras haya colisión entre los dos procesos: en el momento en que el proceso perseguido se distancie un poco del cuadrado blanco de manera que deje de haber colisión, el comando **collision()** volverá a devolver 0, y *flagseguir* mantendrá su valor a 1 igual -se supone que esto nos interesa para seguir manifestando que ha habido colisión y por tanto ha de comenzar la persecución-. El problema viene cuando se quieren evaluar las condiciones *if(x+20<idprota.x)* y similares: cuando se deja de colisionar y por tanto ha de empezar la persecución, acabamos de decir que **collision()** retorna otra vez 0, con lo que *idprota* es 0, con lo que al intentar evaluar estos cuatro ifs problemáticos -ya que *flagseguir* recordemos que vale 1- nos encontramos con que no tenemos proceso a perseguir: el identificador del mismo es 0, o lo que es igual, ningún proceso. Así que no se puede perseguir a nadie y se produce el error.

Imaginémosnos por último que en pantalla hay más procesos susceptibles de colisionar con el cuadrado blanco y ser perseguidos por éste (un cuadrado verde, otro azul,etc). Si tenemos el código anterior, modificarlo para que se adapte a estas nuevas circunstancias es un juego de niños.

```
import "mod_key";
import "mod_proc";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_rand";
import "mod_video";
```

```
process main()
begin
  set_mode(640,480,32);
  perseguido1();
  perseguido2();
  perseguidor();
  repeat
     frame;
  until(key(_esc))
  let_me_alone();
end

process perseguido1()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
  x=300;y=300;
  loop
     if(key(_left)) x=x-10;end
     if(key(_right))x=x+10;end
     if(key(_up))   y=y-10;end
     if(key(_down)) y=y+10;end
     frame;
  end
end

process perseguido2()
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,255));
  x=100;y=100;
  loop
     if(key(_a)) x=x-10;end
     if(key(_d)) x=x+10;end
     if(key(_w)) y=y-10;end
     if(key(_s)) y=y+10;end
     frame;
  end
end

process perseguidor()
private
   int idprota;
   int flagseguir=0;
end
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
  x=200;y=200;
  loop
       if(idprota=collision(type perseguido1))
                     flagseguir=idprota;
       elif (idprota=collision(type perseguido2))
                     flagseguir=idprota;
       end
/*Los +20 y -20 son para que el perseguidor no se coloque encima del perseguido, sino que
se quede al lado.Estos ifs lo que hacen es comprobar si el perseguido está a la
derecha/izquierda/arriba/abajo respecto el perseguido, para moverse en consecuencia.
Fijarse que el perseguidor va un poco más lento que el  perseguido, para causar un efecto
chulo.*/
      if (flagseguir!=0)
            if (x+20<flagseguir.x)x=x+7;end
            if (x-20>flagseguir.x)x=x-7;end
            if (y+20<flagseguir.y)y=y+7;end
            if (y-20>flagseguir.y)y=y-7;end
       end
       frame;
   end
end
```

Lo único que se ha hecho (aparte de incluir un nuevo proceso perseguido: un cuadrado rosa) es incluir la

posibilidad, dentro del código del proceso perseguidor, de colisionar o bien con un proceso de tipo “perseguido1()” o bien “perseguido2()”. Y ya está: todo lo demás permanece igual.

Comentar finalmente que, si pruebas este último código, verás que el cuadrado blanco es muy “promiscuo”. Si está persiguiendo un determinado proceso y en un momento dado colisiona con otro proceso diferente, el perseguidor abandona a su antiguo perseguido para comenzar a perseguir al que acaba de colisionar con él. ¿Se te ocurre alguna manera evitar esto, es decir, para hacer que una vez el proceso perseguidor haya empezado a perseguir a otro proceso, ya no se cambie el proceso perseguido por ningún otro más?

Otro ejemplo más sofisticado de persecución de procesos (donde sí utilizamos funciones como **get_angle(),get_dist(),get_distx()** o **get_disty()**) es el siguiente. Si lo ejecutas verás que tienes un cuadrado verde (el “protagonista”) que lo podrás mover con los cursores, y tres cuadrados rojos (los “enemigos”) que harán dos cosas: por un lado siempre estarán encarados al protagonista: es decir, su ángulo será tal que en cada movimiento que haga el protagonista, los enemigos irán actualizando su orientación para “vigilarlo” permanentemente. Este efecto, aunque no lo parezca, en realidad no es necesario para lo siguiente. Y lo siguiente es que, si el protagonista se acerca demasiado a un enemigo (ya que éste tiene definida una zona de “sensibilidad”), el enemigo se “enterará” de la presencia del protagonista y éste será perseguido con insistencia hasta que se aleje lo suficiente para salir de esa zona de “sensibilidad”, momento en el que el enemigo dejará de “notar” que hay alguien cerca y se volverá a quedar quieto. Una aplicación directa de este código la veremos en el capítulo-tutorial sobre RPG.

```
import "mod_key";
import "mod_proc";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";

process main()
begin
set_mode(640,480);
protagonista(320,240);
enemigo(100,200);
enemigo(50,100);
enemigo(500,300);
loop
     frame;
     if(key(_esc)) exit();end
end
end

PROCESS protagonista(x,y)
BEGIN
graph=new_map(30,30,32);map_clear(0,graph,rgb(0,255,0));
loop
  if(key(_left)) x=x-4;end
  if(key(_right))x=x+4;end
  if(key(_up)) y=y-4;end
  if(key(_down)) y=y+4;end
  frame;
end
END

Process enemigo(x,y)
private
          int idprota;
          int angleprota;
           int distprota;
          int move_x,move_y;
end
Begin
graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
Loop
//Obtengo el identificador del proceso protagonista (sólo hay uno)
    idprota=get_id(type protagonista);
//Obtengo el ángulo entre el proceso enemigo actual y el protagonista, para...
```

```
    angleprota=get_angle(idprota);
/*...hacer que el enemigo esté siempre "mirando" al protagonista. Ésta es la línea que lo
hace posible: con un poco de geometría en un papel podrás ver de dónde sale la fórmula:
sólo hay que tener claro qué ángulo representa "angleprota" y qué ángulo es "angle".*/
    angle=angleprota-90000;
//Obtengo la distancia entre el proceso enemigo actual y el protagonista
    distprota=get_dist(idprota);

/*2ª parte: Si el protagonista se acerca demasiado al enemigo, el cual siempre lo estará
vigilando, quiero que éste sea perseguido. Para ello podría aprovechar la función
get_dist utilizada en la línea anterior, pero se va a hacer de otra manera, por
componentes. Primero obtengo las componentes X e Y de la distancia al protagonista,
para...*/
     move_x=get_distx(angleprota,distprota);
     move_y=get_disty(angleprota,distprota);

/*..comprobar unas condiciones determinadascon estos cuatro ifs. Estas condiciones
provocan que, si las componentes X e Y de la distancia están dentro de un rango
determinado, (es decir, si el protagonista está demasiado cerca del enemigo), éste se
mueva en una determinada dirección,que es la dirección donde se supone está el
protagonista. Las condiciones de los ifs se pueden ver de dónde salen si coges un papel y
un lápiz y dibujas estas cuatro condiciones posibles. El rango -los números de las
condiciones del if- se puede cambiar para hacer que el enemigo sea más o menos sensible a
la presencia del protagonista. */

/*Esta condición ocurrirá cuando el protagonista está a la izquierda del enemigo (entre
20 y 100 píxeles horizontalmente) y a su misma coordenada vertical más/menos 100
píxeles*/
     if ((move_x<-20 and move_x>-100) and (move_y>-100 and move_y<100))
/*Notar que el desplazamiento del enemigo no lo hago en función de su ángulo, sino en
base a movimientos horizontales o verticales. Lo de las líneas anteriores que hacían que
el enemigo estuviera encarando al protagonista si éste se acercaba demasiado es un efecto
puramente estético*/
          x=x-1;
    end
/*Esta condición ocurrirá cuando el protagonista está a la derecha del enemigo (entre 20
y 100 píxeles horizontalmente) y a su misma coordenada vertical más/menos 100 píxeles.*/
      if ((move_x>20 and move_x<100) and (move_y>-100 and move_y<100)) x=x+1;  end
/*Esta condición ocurrirá cuando el protagonista está por arriba del enemigo (entre 20 y
100 píxeles verticalmente) y a su misma coordenada horizontal más/menos 100 píxeles.*/
    if ((move_y<-20 and move_y>-100) and (move_x>-100 and move_x<100))  y=y-1; end
/*Esta condición ocurrirá cuando el protagonista está por debajo del enemigo (entre 20 y
100 píxeles verticalmente) y a su misma coordenada horizontal más/menos 100 píxeles.*/
   if ((move_y>20 and move_y<100) and (move_x>-100 and move_x<100))   y=y+1; end
   Frame;
end //loop
END
```

En el ejemplo anterior hemos visto, entre otras cosas, cómo hacer para que un proceso encare a otro permanentemente (es decir, que un proceso varíe su ángulo según se mueva otro proceso, apuntándolo permenentemente, como por ejemplo cuando un enemigo vigila sin descanso al protagonista). De todas maneras, tal como lo hemos hecho, es posible que no nos venga bien en determinadas circunstancias. Imagínate que en vez de cuadrados estamos trabajando con gráficos de personajes: el cuadrado verde es nuestro héroe valeroso y los cuadrados rojos en realidad son feroces orcos, con un gráfico asociado tal como éste:

Para que veas lo que pretendo decirte, cambia el código anterior para que en vez de tener cuadrados rojos, tengas el gráfico anterior o cualquier otro. Yo a este gráfico lo he llamado "orco.png".Te vuelvo a poner el

código haciendo este cambio (quitando las líneas que no nos interesan: ahora no queremos perseguir al protagonista):

```
import "mod_key";
import "mod_proc";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";

Global
      int idorco,idorco2,idorco3;
End

process main()
begin
set_mode(640,480);
idorco=load_png("orco.png");
idorco2=load_png("orco2.png");
idorco3=load_png("orco3.png");
protagonista(320,240);
enemigo(100,200);
enemigo(50,100);
enemigo(500,300);
loop
     frame;
     if(key(_esc)) exit();end
end
END

PROCESS protagonista(x,y)
BEGIN
graph=new_map(30,30,32);map_clear(0,graph,rgb(0,255,0));
loop
  if(key(_left)) x=x-4;end
  if(key(_right))x=x+4;end
  if(key(_up)) y=y-4;end
  if(key(_down)) y=y+4;end
  frame;
end
END

Process enemigo(x,y)
private
          int idprota;
          int angleprota;
end
Begin
graph=idorco;
Loop
    idprota=get_id(type protagonista);
    angleprota=get_angle(idprota);
    angle=angleprota-90000;
    frame;
end
end
```

Si pruebas este código verás lo que ocurre. El gráfico de los orcos cambia su orientación según la posición del cuadrado verde, tal como esperábamos, pero el efecto ¡queda fatal! Los orcos parecen astronautas que están flotando a expensas de nuestro protagonista. ¿Cómo podemos hacer que, en vez de esto, lo que ocurra es que si se detecta que el protagonista está a un cierto ángulo del orco, el gráfico de éste pase automáticamente a ser otro, como éste, llamado "orco2.png":

si el héroe vaga por su derecha,o éste otro (llamado “orco3.png”):

si va por su izquierda? Bueno, tendremos que modificar el proceso “enemigo()” de tal manera que quede así (los cambios están marcados en negrita):

```
Process enemigo(x,y)
private
          int idprota;
          int angleprota;
          int pepe;
end
Begin
graph=idorco;
Loop
    idprota=get_id(type protagonista);
    angleprota=get_angle(idprota);
    pepe=angleprota-90000;

/*Piensa que "pepe" tiene el siguiente rango de valores: vale desde 0 en los puntos que
están alineados justo en la linea vertical centrada en el enemigo pero por encima de
éste, hasta 180 en los puntos de la misma línea pero por debajo del enemigo siempre que
el proceso protagonista esté a la izquierda del enemigo. Si el protagonista está a la
derecha del enemigo, pepe valdrá desde 0 hasta -179 yendo desde la perpendicular de
encima del enemigo hasta la de debajo suyo.. Esto lo puedes ver si sólo tienes un proceso
enemigo funcionando e incluyes fuera de este loop la línea write_var(0,100,100,4,pepe);
para ver los valores de pepe según la posición del protagonista, y tener así una guía
para construir las condiciones de los ifs siguientes.*/
    if(pepe>45000 and pepe<135000)flags=0;graph=idorco3; end //izquierda
    if(pepe>-135000 and pepe<-45000)flags=0;graph=idorco2;end //derecha
    if((pepe>0 and pepe<45000)or(pepe<-1 and pepe>-45000))flags=2;graph=idorco;end //arriba
    if((pepe>135000 and pepe< 180000) or (pepe<-135000 and pepe>-179000))
flags=0;graph=idorco; end //abajo
    frame;
end
end
```

Lo que hemos hecho ha sido cambiar la antigua línea que alteraba la orientación de los enemigos (ya que cambiaba su ***ANGLE***) por otra muy similar donde el ángulo corregido que nos da la posición relativa del protagonista respecto el enemigo lo almacenamos en una variable privada que la hemos llamado *pepe*. Así, para empezar,los enemigos no cambiarán la orientación de su gráfico, pero continuarán sabiendo dónde está el protagonista en cada momento, gracias a *pepe*. La idea es, que según en qué zona respecto el enemigo se sitúe el protagonista (zona superior, derecha, izquierda, inferior), el enemigo cambiará su gráfico por uno acorde, dando la sensación de ojo avizor permanente, ya que los enemigos estarán permanentemente mirando al protagonista, esté donde esté.. Por último, comentar que está claro que nos podríamos haber ahorrado el gráfico “orco3.png” jugando con los flags (al igual que hemos hecho cuando el protagonista está arriba o abajo del enemigo).

Pasemos a otro tema. En ejemplos anteriores ya hemos podido ver que uno de los problemas típicos que tendremos que solventar a menudo en nuestros juegos es averiguar la distancia mínima entre diferentes procesos. Para resolver esta necesidad ya sabes que contamos con la función **get_dist()** y sus “parientes” **get_distx()** y **get_disty()**. Un caso típico es el de los juegos de estrategias, donde los escuadrones tienen que decidir a qué enemigo atacar y

deciden atacar al más cercano. Por eso se va a presentar a continuación algonos ejemplos de código donde hay un proceso que detecta al más cercano de todos sus "enemigos".

Para empezar, pondremos un código que muestra por pantalla un cuadrado blanco en la esquina inferior derecha de la pantalla (nuestro proceso "prota()") y 400 cuadrados azules repartidos aleatoriamente por toda la pantalla (procesos "enemigo()"). El programa lo primero que hará será obtener todas las distancias de cada enemigo con el proceso "prota()" e seguidamente irá matando uno a uno todos los procesos enemigos, por orden de cercanía al "prota()". Es decir, primero morirá (y por tanto desaparecerá de la pantalla) el enemigo más cercano al cuadrado blanco,después el segundo más cercano,etc hasta llegar al último enemigo, el más lejano. El efecto resultante es como si hubiera una onda expansiva en forma de círculo que se engrandece y que arrasa los cuadrados azules.

```
import "mod_key";
import "mod_proc";
import "mod_grproc";
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";
import "mod_rand";
import "mod_text";

global
          int idenemigo[399];
end

process main()
private
          int i;
end
begin
  set_mode(640,480);
  for(i=0;i<400;i++)
      idenemigo[i]=enemigo(rand(10,600),rand(10,400));
//    frame; //Se puede descomentar para ver la creación de los enemigos en tiempo real
  end
/*Importante crear el protagonista después de los enemigos, ya que en su código se
calcula las distancias con éstos:si los enemigos todavía no existieran el programa se
bloquearía*/
  prota();
  while(!key(_esc)) frame;  end
  exit();
end

process prota()
private
        int i;
          int objetivo;
          int distaux, distminima;
end
begin
  graph=new_map(5,5,32);map_clear(0,graph,rgb(255,255,255));
  x=600;y=450;
  loop
/*Típico algoritmo para encontrar el valor mínimo de una lista de valores: se va
recorriendo los valores del vector y si se encuentra un valor menor al que hasta ahora
era el mínimo, este valor pasa a ser el nuevo mínimo.Se empieza por un valor mínimo muy
alto para que el primer elemento sea siempre el mínimo y se puedan hacer las
comparaciones.*/
    distminima=10000;
    for(i=0;i<400;i++)
          distaux=get_dist(idenemigo[i]);
/*La condición de "distaux>0" es MUY importante.Ya que cada vez que se mate un enemigo,
se volverá a realizar este algoritmo otra vez  para encontrar al siguiente enemigo más
cercano, el vector idenemigo[i]tendrá cada vez más elementos que corresponderán a
identificadores de procesos ya muertos. Si realizamos un get_dist con un identificador no
existente, get_dist nos desvuelve 0, con lo que siempre tendríamos como distancia mínima
la calculada con procesos inexistentes. Evidentemente, esto está fatal.*/
```

```
            if(distaux<=distminima and distaux>0)
                        distminima=distaux;
                        objetivo=idenemigo[i];
            end
    end
    write_var(0,100,100,3,objetivo); //Vemos el identificador del enemigo a destruir
    signal(objetivo,s_kill);
    frame;
end //loop
end

process enemigo(x,y)
begin
  graph=new_map(5,5,32);map_clear(0,graph,rgb(0,0,255));
  loop
            frame;
  end
end
```

Vamos a jugar con este código un rato. Escribe la siguiente línea: *signal(objetivo,s_kill);* al final del (único) if del proceso "prota()". ¿Qué pasa si ejecutas el programa ahora? Que los enemigos son asesinados a un ritmo mucho más vertiginoso. ¿Podrías explicar el por qué? Porque estamos eliminando todos aquellos procesos que MIENTRAS se está recorriendo el bucle sean designados "objetivo", pero el objetivo correcto sólo se puede saber DESPUES de haber recorrido todo el bucle, que es cuando se sabrá el proceso a la menor distancia de entre todos: si se mata mientras se va recorriendo el bucle, se irán encontrando varios objetivos, uno por cada proceso que tenga una distancia menor que los anteriores, pero que no necesariamente sea la mínima total, porque no se ha acabado de recorrer el bucle entero.

Importante recalcar que las órdenes **signal()** se ejecutarán inmediatamente, sin esperar a ninguna línea Frame;. Esto quiere decir que si mientras estamos dentro del bucle se ejecutan varias órdenes **signal()**, una para cada objetivo encontrado diferente, todas estas órdenes *s_kill* diferentes se ejecutarán inmediatamente, sin esperar a llegar a ningún fotograma. Es por eso que la eliminación es mucho más rápida:porque cuando por fin se llega a un fotograma, ya se han encontrado antes varios objetivos que han sido eliminados.

Ahora vamos a hacer otra cosa. Vamos a hacer que nuestro cuadradito blanco se desplace a la posición del enemigo más cercano, éste se muera y el cuadradito blanco se vuelva a desplazar al siguiente enemigo más cercano, éste se muera, y así. Para ello, lo único que tenemos que hacer es añadir justo después de la línea *write_var(0,100,100,3,objetivo);* del proceso *"prota()",* las líneas siguientes:

```
if(exists(objetivo)==true) x=objetivo.x;end
if(exists(objetivo)==true) y=objetivo.y;end
```

Si ejecutas este proceso verás que ahora ya no aparece el efecto de onda expansiva, sino que como las distancias mínimas se recalculan en cada iteración del LOOP principal, y precisamente en cada una de éstas cambiamos las coordenadas ***X*** e ***Y*** del protagonista por las que en ese momento tiene el objetivo actual -si existe-, lo que obtenemos es nuevas medidas de las distancias en cada nueva posición del cuadradito blanco, con lo que los enemigos que mueren son los que están más cerca del protagonisa en un momento dado.

Otro código muy interesante donde se puede ver la utilidad de la detección de las distancias mínimas entre procesos, y sus orientaciones recíprocas es el siguiente (aunque ya advierto que incluye órdenes no vistas todavía). Este ejemplo representa el combate de dos ejércitos: uno azul y otro rojo. El usuario puede generar soldados azules con el botón derecho del ratón, y los soldados rojos se generarán en posiciones aleatorias cada x tiempo. La gracia del asunto radica en que cada soldado de ambos bandos se irá aproximando a aquel soldado del otro bando (y sólo del otro bando) que tenga más cerca. En el momento que llegue a una distancia mínima, ese soldado se parará para comenzar a dirigir a su enemigo disparos de flechas sin parar. Cuando un soldado recibe un número determinado de éstas, muere y desaparece de pantalla. Además, existe el modo "formación de combate", invocado con la tecla SPACE, el cual consiste en lo mismo pero donde los soldados de ambos bandos se sitúan ordenadamente a cada lado de la ventana en forma de escuadrón listo para la batalla. Lo interesante del ejemplo es sobretodo el sistema de detección del enemigo más cercano (similar al de los ejemplos anteriormente vistos), la aproximación a éste y el direccionamiento hacia éste de las flechas.

```
/*Este tutorial explica cómo se puede detectar la distancia más corta entre todos los
soldados enemigos (cuadrados rojos -bando 2-) y nuestros personajes (cuadrados azules
-bando 1-)*/
import "mod_key";
import "mod_proc";
import "mod_grproc";
```

```
import "mod_math";
import "mod_draw";
import "mod_map";
import "mod_video";
import "mod_rand";
import "mod_text";
import "mod_screen";

global
  int idsold;
  int idobjetivo;
  int grafflecha;
end

local
   int bando;
   int energia;
end

process main()
private
  int i,j;
  int grafsold1;
  int grafsold2;
  int modonormal=1; /*Variable que vale 1 si el juego está en modo "normal" y 0 si está
en modo "Combate en formación" (al pulsar la tecla SPACE)*/
end
BEGIN
 set_mode(640,480);
 set_fps(30,0);
 write(0,10,10,3,"Utilizar el ratón para poner tus personajes y testear las distancias");
//Gráfico de tus personajes
 grafsold1=new_map(10,10,32);map_clear(0,grafsold1,rgb(0,0,255));
//Gráfico de tus enemigos
  grafsold2=new_map(10,10,32);map_clear(0,grafsold2,rgb(255,0,0));
//Gráfico de la flecha
  grafflecha=new_map(5,5,32);drawing_map(0,grafflecha);draw_box(0,0,5,2);
  mouse.graph=grafsold1;mouse.flags=4;
  /*Aparición de 5 soldados enemigos y 5 propios no importa dónde*/
   FROM i=0 TO 5;
//Soldado bando 1 (nuestros personajes)
     soldado(rand(20,620),rand(20,380),grafsold1,1);
//Soldado bando 2 (nuestros enemigos)
     soldado(rand(20,620),rand(20,380),grafsold2,2);
   END
   LOOP
//Si se clica el botón derecho del ratón, aparecerá un soldado del bando 1
       IF(mouse.right)soldado(mouse.x,mouse.y,grafsold1,1); END
/*Modo combate en formación. Se eliminan los soldados que estuvieran en este momento y se
generan nuevos  soldados colocados en formación*/
      IF(key(_space))
          IF(modonormal==1) //Para no hacer un modo formación si ya hay uno funcionando
              signal(TYPE soldado,s_kill);
              FROM i=10 TO 80 STEP 10;
                    FROM j=10 TO 390 STEP 20;
                            soldado(i,j,grafsold1,1);
                            soldado(640-i,j,grafsold2,2);
                     END
              END
/*Después de haber creado la formación, vuelvo al estado "normal", donde los enemigos se
crean en posiciones aleatorias*/
          modonormal=0;
          END
      ELSE
          modonormal=1;
          //Aparición de los soldados del bando 2 aleatoriamente
          IF(rand(0,100)<5)soldado(rand(20,620),rand(20,380),grafsold2,2);END
      END
```

```
      IF(key(_esc)) exit(); END
      FRAME;
   END //Loop
END

/*Proceso de los soldados de los dos campos*/
PROCESS soldado(x,y,graph,int bando)
private
   int tiro;
   int dist;
end
BEGIN
   energia=50;
   LOOP
//Detección del soldado enemigo (entre sí) más cercano (el algoritmo ya es conocido)
      idobjetivo=0;
      dist=1000;
      WHILE(idsold=get_id(TYPE soldado)) //Mira todos los procesos soldados
//Si el bando del proceso soldado es diferente de este soldado...
          IF(idsold.bando<>bando)
//...detecta si la distancia entre el soldado es inferior a la distancia guardada antes
               IF(get_dist(idsold)<dist)
                  dist=get_dist(idsold); //Si es inferior, guarda la distancia actual
                  idobjetivo=idsold;     //El objetivo a aniquilar será ese soldado
               END
          END
      END

//Si hay un objetivo fijado, después de haber pasado por el bucle anterior...
      IF(idobjetivo<>0)
//..y si la distancia entre el soldado y el objetivo es < 100, le lanzo una flecha
            IF(get_dist(idobjetivo)<200)
                  tiro=tiro+1;
/*Este if es simplemente para reducir la frecuencia con el que se lanzarán las flechas,
ya que si no se pusiera se estarían generando flechas a cada frame. La restricción por un
lado es hacer que un número aleatorio sea menor que una cantidad dada (sistema visto en
el capítulo anterior del tutorial del matamarcianos), pero además se ha de cumplir una
condición más que es que la variable tiro (la cual va aumentando su valor a cada frame en
la línea anterior) ha de ser mayor de 15*/
                   IF(rand(0,100)<10 AND tiro>15)
                                 tiro=0; //Se vuelve a resetear "tiro"
/*La flecha sale dirigida en dirección al objetivo. El 10 que se multiplica es el valor
de la "resolution" del proceso flecha.*/
                                 flecha(x*10,y*10,get_angle(idobjetivo),grafflecha);
                   END
/*Si la distancia es mayor, en vez de lanzar una flecha, me muevo hacia él, de tal manera
que llegará un momento en que la distancia ya sí sea menor y pueda entonces pararme y
proceder al disparo*/
            ELSE
                    x=x+get_distx(get_angle(idobjetivo),3);
                    y=y+get_disty(get_angle(idobjetivo),3);
          END
      END

      IF(energia<=0) signal(id,s_kill); END //Si no tengo energía, muero
      FRAME;
   END
END

/*Las flechas*/
PROCESS flecha(x,y,angle,graph)
Private
    int vidaflecha;
end
BEGIN
  bando=father.bando; //La flecha será del bando de su padre
  resolution=10; //Para que las flechas salgan hacia el objetivo de forma más precisa
  LOOP
```

```
            //A cada frame, la flecha avanza y aumenta su tiempo de vida
           advance(5*resolution);vidaflecha=vidaflecha+1;
           //Si la flecha choca contra un soldado...
           IF(idsold=collision(TYPE soldado))
             //y si el soldado es diferente de su bando...
            IF(idsold.bando<>bando)
             //...este soldado pierde 10 puntos de energía
                          idsold.energia=idsold.energia-10;signal(id,s_kill);
            END
         END
        //Si la flecha sale de la pantalla o ya ha volado demasiado, muere
         IF(out_region(id,0)OR vidaflecha>50) signal(id,s_kill); END
         FRAME;
    END
END
```

En general, podemos concluir que cuando se tienen muchos procesos de un tipo, como en el caso anterior con los 400 enemigos, es recomendable por claridad y comodidad contar en nuestro programa con un proceso "controlador()", que gestione de forma centralizada la creación/manipulación/destrucción de múltiples procesos. De esta forma se gana en limpieza de código y rapidez en la depuración de errores. Por ejemplo, un proceso controlador tipo sería el siguiente:

```
process controlador ()
private
  int enemigos[los-que-necesites];
end
begin
  enemigos[0]=enemigo();//Creación de múltiples procesos y almacenamiento de su ID en un vector
  enemigos[1]=enemigo();
...
  enemigos[los-que-necesites]=enemigo();

//Bucle para comprobar todos los enemigos
for (cont=0;cont<=los-que-necesites;cont++)
/*Aquí podemos escribir lo que necesitamos. En este caso por ejemplo miramos si un
enemigo concreto (sólo uno cada vez) está dentro de un determinado rango de acción. Si es
así, lo despertamos; si no, lo dormimos.*/
   if (enemigos[cont].x<distancia_minima_x and enemigos[cont].y<distancia_minima_y)
       signal(enemigos[cont],s_wakeup); //Despertamos a ese enemigo
   else //De lo contrario, lo congelamos y le quitamos el grafico
       signal(enemigos[cont],s_freeze); //Puedes usar sleep, que le quita el grafico.
       enemigos[cont].graph=0;
   end
end //for
...
loop
   frame;
end
end
```

| **NEAR_ANGLE(ANGULO1,ANGULO2,MAX_INC)** |
|---|
| Esta función, definida en el módulo "mod_math", suma a ANGULO1 un número de grados dado por MAX_INC y devuelve el nuevo ángulo, siempre y cuando el resultado no sea un ángulo mayor a ANGULO2, en cuyo caso el ángulo devuelto será exactamente ANGULO2.<br><br>Es decir, dicho de otro modo, si la distancia entre los dos ángulos es más pequeña que MAX_INC, entonces el ángulo resultante devuelto será exactamente ANGULO2.<br><br>El valor de MAX_INC ha de ser positivo, para que el proceso de aproximación se realice por el lado de la circunferencia donde la distancia entre los dos ángulos sea más pequeña.<br><br>Esta función sirve de ayuda para hacer que procesos puedan perseguir a otros pero con una capacidad limitada de giro en cada frame. |

PARAMETROS:
INT ANGULO1: Ángulo original en milésimas de grado (90000= 90º)
INT ANGULO2 : Ángulo de destino
INT MAX_INC: Número máximo de milésimas de grado de incremento del ángulo original

Un ejemplo trivial para probar esta función sería algo como éste:

```
import "mod_key";
import "mod_math";
import "mod_map";
import "mod_text";

process main()
private
  int ang=10000,ang2=20000,ang3;
end
begin
  repeat
    delete_text(0);
    ang3=near_angle(ang,ang2,9999);
    write(0,100,100,4,ang);
    write(0,100,110,4,ang2);
    write(0,100,120,4,ang3);
    frame;
  until(key(_esc))
end
```

donde se puede ir cambiando el valor del tercer parámetro de **near_angle()** para probar los efectos. Otro ejemplo más útil es el siguiente:

```
import "mod_key";
import "mod_math";
import "mod_map";
import "mod_text";
import "mod_map";
import "mod_mouse";
import "mod_grproc";

process main()
private
  int ang;
end
begin
  graph=new_map(10,10,32);
  map_clear(0,graph,rgb(0,0,255));
  mouse.graph=new_map(10,10,32);
  map_clear(0,mouse.graph,rgb(255,255,0));
  repeat
    ang=fget_angle(x,y,mouse.x,mouse.y);
    angle=near_angle(angle,ang,10000);
    advance(5);
    frame;
  until(key(_esc))
end
```

¿No ves lo que pasa? Sustituye la línea donde aparece **near_angle()** por ésta otra: *angle=ang* ;. Comprueba qué pasa. ¿Podrías explicar por qué? Y otro ejemplo:

```
import "mod_key";
import "mod_math";
import "mod_map";
import "mod_text";
import "mod_map";
import "mod_mouse";
import "mod_grproc";
import "mod_video";
```

```
import "mod_draw";
import "mod_rand";
import "mod_proc";

local
   int count=0; //Cada proceso target tiene su contador de impactos
end

process main()
 private
   int graphic; //Para el gráfico del cursor del ratón
 end
 begin
   set_fps(30,0);
   target(50,50); //4 objetivos inmóviles que recibirán el impacto de los misiles
   target(250,70);
   target(70,200);
   target(280,180);
   graphic=draw_circle(x,y,2); //El cursor del ratón es una primitiva
   loop
     move_draw(graphic,mouse.x,mouse.y);
     if(mouse.left)missile(mouse.x,mouse.y);end //Los misiles partirán del cursor
     if(key(_esc)) exit(); end
     frame;
   end
 end

process target(x,y)
 private
   int dist; //Distancia de cada objetivo al cursor del ratón
 end
 begin
   write_var(0,x,y-5,4,count); //Muestro el número de impactos que lleva recibidos
   write_var(0,x,y+7,4,dist); //Muestro la distancia al cursor del ratón en tiempo real
   //Los objetivos son simples pixels blancos
   put_pixel(x,y,65535);
   put_pixel(x+1,y,65535);
   put_pixel(x,y+1,65535);
   put_pixel(x+1,y+1,65535);
   loop
     dist=fget_dist(mouse.x,mouse.y,x,y); //Distancia al cursor del ratón
     frame;
   end
 end

process missile(x,y)
 private
   int mindist;
   int idmascercano;
   int tempdist;
   int tempid;
 end
 begin
   angle=1000*rand(70,110); //Ángulo inicial del misil es aleatorio
   loop
/*Borro el rastro actual del misil en el fotograma anterior (de hecho, lo que hago es
pintarlo de negro -o sea, transparente-), antes de que avance una nueva posición */
     put_pixel(x,y,rgb(0,0,0));
     put_pixel(x+1,y,rgb(0,0,0));
     put_pixel(x,y+1,rgb(0,0,0));
     put_pixel(x+1,y+1,rgb(0,0,0));

/*Vuelvo a usar el mismo algoritmo de siempre para detectar el objetivo más cercano
-distancia mínima- en cada frame respecto el misil actual.Al final obtengo "idmascercano"
(que representa el id de uno de los cuatro objetivos) y "mindist",(que es la distancia de
éste respecto el misil)*/
     mindist=10000;
     idmascercano=0;
```

```
     while(tempid=get_id(type target))
        if((tempdist=fget_dist(x,y,tempid.x,tempid.y))<mindist)
           idmascercano=tempid;
           mindist=tempdist;
         end
     end

     //Si se encuentra un objetivo cercano...
     if(exists(idmascercano))
 /*...y la distancia de éste respecto el misil es menor que una dada, aumento el contador
de ese objetivo, y mato el misil correspondiente*/
           if(mindist<8)idmascercano.count++;return;end
//El misil varía su orientación para teledirigirse de forma espiral sobre algún objetivo
          angle=near_angle(angle,fget_angle(x,y,idmascercano.x,idmascercano.y),7500);
    end
    advance(9);

    //El misil no es más que simples pixels
    put_pixel(x,y,rgb(255,0,0));
    put_pixel(x+1,y,rgb(255,0,0));
    put_pixel(x,y+1,rgb(255,0,0));
    put_pixel(x+1,y+1,rgb(255,0,0));
    frame;
   end //loop
end
```

## *Trabajar con fades y efectos de color

**FADE(R,G,B,Velocidad)**

Esta función, definida en el módulo "mod_map", activa una transición de colores automática, que puede usarse para oscurecer la pantalla, hacer un fundido, o colorear su contenido.

Llamar a esta función simplemente activa ciertas variables internas y pone a 1 la variable global predefinida ***FADING***. A partir de entonces, el motor gráfico se encargará automáticamente de colorear la pantalla cada frame.

Los valores de componentes R, G y B recibidos por esta función se dan en porcentaje. Un porcentaje de 100% dejará un color sin modificar, mientras un porcentaje de 0% eliminará por completo la componente. También es posible especificar un valor por encima de 100. Se considera que un valor de 200 en una componente, debe poner al máximo la intensidad de la misma.

Estos valores permiten hacer combinaciones flexibles, como por ejemplo un fundido al blanco (200,200,200) o bien aumentar la intensidad de color de una o varias componentes (150,150,100). Tal vez el valor más habitual sea el fundido al negro (0,0,0) o restaurar el estado normal de los colores (100,100,100).

El valor de velocidad es un valor entero entre 1 y 64. Un valor de 64 activaría los porcentajes de color indicados de forma inmediata en el próximo frame, mientras un valor de 1 iría modificando las componentes de forma sucesiva, de forma que alcanzarían los valores especificados como parámetro en aproximadamente unos 60 frames. Un valor intermedio permite realizar la transición más rápidamente.

La variable global ***FADING*** estará a 1 inmediatamente después de llamar a esta función. El motor gráfico la pondrá a 0 cuando la transición alcance los colores especificados como parámetro.

PARAMETROS:

INT R: Cantidad de componente Rojo, en tanto por ciento
INT G: Cantidad de componente Verde, en tanto por ciento
INT B: Cantidad de componente Azul, en tanto por ciento
INT Velocidad: Velocidad

Un ejemplo de esta función podría ser éste:

```
import "mod_key";
```

```
import "mod_map";
import "mod_text";
import "mod_video";
import "mod_draw";
import "mod_screen";
import "mod_proc";

process main()
private
  int idpng;
  int r=100,g=100,b=100,s=3;
end
begin
  idpng=new_map(320,240,32);map_clear(0,idpng,rgb(235,136,98));
  put_screen(0,idpng);
  write(0,10,60,3,"1: Fundido en blanco (fade 200,200,200):");
  write(0,10,70,3,"2: Fundido en negro (fade 0,0,0):");
  write(0,10,80,3,"3: Fundido personalizado (fade %R,%G,%B):");
  write(0,20,90,3,"R: Nuevo valor de R (en porcentaje): ");
  write_var(0,250,90,3,r);
  write(0,20,100,3,"G: Nuevo valor de G (en porcentaje): ");
  write_var(0,250,100,3,g);
  write(0,20,110,3,"B: Nuevo valor de B (en porcentaje): ");
  write_var(0,250,110,3,b);
  write(0,10,140,3,"S: Nuevo valor de Velocidad: ");
  write_var(0,190,140,3,s);
  repeat
    if(key(_1))
      fade(255,255,255,s);
      while(fading==1)
           frame;
           if(key(_esc)) exit(); end
      end
    end
    if(key(_2))
      fade(0,0,0,s);
      while(fading==1)
           frame;
           if(key(_esc)) exit(); end
      end
    end
    if(key(_3))
      fade(r,g,b,s);
      while(fading==1)
           frame;
           if(key(_esc)) exit(); end
      end
    end
    if(key(_r)) r++; if(r>200) r=0; end end
    if(key(_g)) g++; if(g>200) g=0; end end
    if(key(_b)) b++; if(b>200) b=0; end end
    if(key(_s)) s++; if(s>63)  s=1; end end
    frame;
  until(key(_esc))
end
```

El funcionamiento del ejemplo es el siguiente: si se apreta la tecla 1, se realiza un fundido en blanco; si se pulsa la tecla 2, se realiza un fundido en negro; si se apreta la tecla 3, se realiza un fundido partiendo de los valores en porcentaje actuales de las tres componentes, los cuales se pueden aumentar con las teclas “r”,”g” y “b” respectivamente. Para cualquiera de los tres fades posibles, se aplicará la velocidad actual de fundido, la cual se puede aumentar con la tecla “s”.

Fíjate que es muy importante el bucle *while(fading==1) frame; end*, porque es el que posibilita que mientras esté durando el fundido éste se pueda visualizar en pantalla, puesto que obliga a pasar fotogramas sin hacer otra cosa hasta que el fundido se acabe. Si no se hiciera así, el fundido se estaría realizando pero hasta que no se encontrara una orden frame no podría mostrarse su estado en ese momento.

Fíjate también que se ha hecho que se pueda salir del programa incluso en mitad de un fundido, apretando la tecla ESC.

| **FADE_ON()** |
|---|
| Esta función, definida en el módulo "mod_map", equivale exactamente a ejecutar un **fade(100,100,100,16)**.<br><br>Se usa para restaurar la pantalla después de una transición de color como bien puede ser un fundido al negro. La pantalla se restaurará, tras llamar a esta función, en unos 4 frames. |

| **FADE_OFF()** |
|---|
| Esta función es parecida a ejecutar una orden **fade(0,0,0,16)**. Sin embargo, a diferencia de la orden **fade()**, que vuelve inmediatamente, esta función espera a que el fundido al negro termine antes de volver.<br><br>Mientras dure este fundido al negro especial, todos los procesos estarán detenidos así como cualquier tipo de proceso interno, incluyendo la actualización de variables globales. La función tardará aproximadamente unos cuatro frames en terminar. |

Un ejemplo de esta función y la anterior podría ser éste:

```
import "mod_key";
import "mod_map";
import "mod_text";
import "mod_screen";

process main()
private
  int idpng;
  int fades=1;
end
begin
  idpng=new_map(320,240,32);map_clear(0,idpng,rgb(235,136,98));
  write(0,10,90,3,"1: Fade on");
  write(0,10,110,3,"2: Fade off");
  put_screen(0,idpng);
  repeat
    if(key(_1) and fades==2) fade_on();  fades=1; end
    if(key(_2) and fades==1) fade_off(); fades=2; end
    frame;
  until(key(_esc))
end
```

La utilidad más evidente de los fades está en implementarlos en los menús iniciales y las introducciones de los juegos, o para mostrar una pantalla de cambio de nivel,etc.A continuación se presenta un ejemplo donde primeramente aparece un gráfico de fondo (junto con más cosas: un proceso, un texto,etc); al cabo de dos segundos se realiza un fadeoff a negro, eliminando a la vez todo lo demás (los procesos, los textos, el fondo definido); y al cabo de dos segundos se vuelve a hacer un fadein mostrando un gráfico de fondo diferente al original, y sin rastro de procesos,textos,etc.

```
import "mod_key";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_video";
import "mod_timers";
import "mod_screen";

global
          int idpng1,idpng2;
          int hora;
          int espera=200;
end
```

```
process main()
begin
  set_mode(640,480);
  idpng1=new_map(640,468,32);map_clear(0,idpng1,rgb(255,0,0));
  idpng2=new_map(640,468,32);map_clear(0,idpng2,rgb(0,255,255));
  put_screen(0,idpng1); //Mostramos el primer fondo
  proceso(); //Ponemos un proceso pululando por ahí para meter más cosas
  write(0,100,100,4,"HOLAHOLAHOLAHOLAHOLA");//Además escribimos un texto cualquiera

  //Permanecemos un tiempo de espera -2 segundos- viendo todo esto
  hora = timer[0]; //Hora guarda la hora actual
/*Espera es el tiempo (en centésimas de segundo) que queremos que el juego se quede
parado mostrando el fondo, el proceso y el texto*/
  while(timer[0] < hora+espera) frame; end

  fade(0,0,0,1); while(fading==1) frame; end //Visualizamos en fadeoff completo
  let_me_alone(); //Sólo si queremos matar al resto de procesos
  delete_text(all_text); //Borramos todos los textos de pantalla
  clear_screen(); //Borramos todos los fondos que hayamos puesto

  //Permanecemos un tiempo de espera con la pantalla en negro
  hora = timer[0];
  while(timer[0] < hora+espera) frame; end

  //Visualizamos el fadein completo con un nuevo fondo definido justo después
  fade(100,100,100,1); put_screen(0,idpng2); while (fading==1) frame; end

  repeat
          frame;
  until(key(_esc))
  let_me_alone();
end

process proceso()
begin
  graph=new_map(40,40,32);map_clear(0,graph,rgb(200,100,40));
  x=300;y=200;
          loop
                    frame;
          end
end
```

Volver a insistir en la importancia de introducir el bucle *while(fading==1)frame;end* para lograr visualizar el difuminado mientras éste esté activo.

Otra función interesante es la siguiente:

| **GET_RGBA( COLOR, &R,&G,&B,&ALPHA)** |
|---|
| Esta función, definida en el módulo "mod_map", permite obtener las componentes por separado de un color obtenido mediante la función **rgb()** o bien leído de los pixels de un gráfico mediante una función como **map_get_pixel()**.<br><br>Las componentes contendrán valores entre 0 y 255, pero el mismo pixel del mismo gráfico puede devolver valores de componentes diferentes en ordenadores distintos. Es importante no comparar los valores devueltos por **get_rgb()** con ningún valor constante. Sin embargo, se consideran válidos los siguientes usos con los valores devueltos por **get_rgb()**:<br>• Comprobar que una componente está a cero (== 0). Este valor se considera seguro. Es posible comprobar que un color sea el negro puro observando si sus tres componentes valen cero, o bien que sea transparente total observando si sus cuatro componentes valen cero, por ejemplo.<br>• Comparar componentes de dos valores obtenidos mediante **rgb(), map_get_pixel()** u otra función que trabaje con colores, durante la misma ejecución del programa.<br>• Comparar valores de forma aproximada (por ejemplo, comparar si una componente tiene un valor mayor a |

128) cuando no sea importante obtener un resultado exacto ni el mismo en todos los ordenadores.

PARAMETROS:

INT Color: Un color de 16 bits
POINTER R: Variable de tipo INT que contendrá la componente roja
POINTER G: Variable de tipo INT que contendrá la componente verde
POINTER B: Variable de tipo INT que contendrá la componente azul
POINTER ALPHA : Variable de tipo INT que contendrá el nivel de transparencia

Existe otra función llamada **get_rgb()** que funciona exactamente igual salvo por el detalle de que carece del último parámetro de **get_rgba()** (la componente alpha). Por lo tanto, no puede trabajar con transparencias, así que su uso está más enfocado a modos de vídeo de 8 ó 16 bits.

Un ejemplo de esta función podría ser:

```
import "mod_key";
import "mod_map";
import "mod_text";
import "mod_timers";
import "mod_rand";

process main()
private
  int Color;
  int r,g,b,a;
end
begin
  write(0,10,85,3,"Número de color: ");
  write_var(0,125,85,3,Color);
  write(0,10,95,3,"Valor R: ");
  write_var(0,125,95,3,r);
  write(0,10,105,3,"Valor G: ");
  write_var(0,125,105,3,g);
  write(0,10,115,3,"Valor B: ");
  write_var(0,125,115,3,b);
  write(0,10,125,3,"Valor A: ");
  write_var(0,125,125,3,a);
  timer[0]=200;
  repeat
    if(timer[0]>200)
    /*Aquí el color se genera automáticamente pero también es normal obtener
    el valor de color a partir de la función map_get_pixel()*/
      color=rgba(rand(0,255),rand(0,255),rand(0,255),rand(0,255));
      get_rgba(Color,&r,&g,&b,&a);
      timer[0]=0;
    end
    frame;
  until(key(_esc))
end
```

Y finalmente, una serie de funciones que aplican diversos efectos -difuminado, filtrado, etc- a imágenes (aunque sólo funcionan en profundidad de 16 bits).

**BLUR(LIBRERIA,GRAFICO, MODO)**

Esta función, definida en el módulo "mod_effects", difumina un gráfico según un modo determinado. Sólo funciona en modo de vídeo de 16 bits (y con imágenes de dicha profundidad).

El modo puede ser uno de los siguientes:

- 0: Solo se tienen en cuenta los pixels situados a la izquierda y arriba de cada pixel.
- 1: 3x3. Se tienen en cuenta todos los pixels colindantes (8).
- 2: 5x5. Se tienen en cuenta los 24 pixels que rodean cada pixel.
- 3: 5x5 con mapa adicional. Igual que el anterior pero usando un mapa temporal.

El modo define la calidad y la velocidad del difuminado. Un número mayor de modo tiene más calidad, pero requiere mayor tiempo de proceso.

PARAMETROS:
INT LIBRERIA: Identificador de la librería cargada con **load_fpg()**, ó 0 si no es el caso.
INT GRAFICO: Número de gráfico dentro de esa librería, o identificador de la imagen si se cargó independiente
BYTE MODO: Modo de difuminación

Un ejemplo de esta función, que creo que no necesita explicación, es el siguiente:

```
import "mod_key";
import "mod_effects";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_video";

global
  int png, png2, png3;
end

process main()
begin
  set_mode(640,480,16);
//Creo una imagen compuesta por tres cuadrados concéntricos
  png=new_map(50,50,16);map_clear(0,png,rgb(255,0,255));
  png2=new_map(30,30,16);map_clear(0,png2,rgb(255,255,0));
  png3=new_map(10,10,16);map_clear(0,png3,rgb(0,255,255));
  map_put(0,png2,png3,15,15);
  map_put(0,png,png2,25,25);

  bola(40,110,"T -1",-1);//No se produce difuminado
  bola(160,110,"T 0",0);
  bola(280,110,"T 1",1);
  bola(400,110,"T 2",2);
  bola(520,110,"T 3",3);
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

process bola(x,y,string txt,int difum)
begin
  write(0,x,y-20,4,txt);
  graph=map_clone(0,png);
  blur(0,graph,difum);
  loop
    frame;
  end
end
```

**FILTER(LIBRERIA,GRAFICO,&VECTOR)**

Esta función, definida en el módulo "mod_effects", aplica un filtro personalizado a un gráfico. Sólo funciona en modo de vídeo de 16 bits (y con imágenes de dicha profundidad).

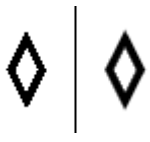
PARAMETROS:
INT LIBRERIA: Identificador de la librería cargada con **load_fpg()** ó 0 si no es el caso.
INT GRAFICO: Número de gráfico dentro de esa librería, o identificador de la imagen si se cargó independiente
POINTER INT VECTOR: Vector de 10 enteros: los 9 primeros definen una matriz de convolución 3x3 y el décimo el factor de escala (normalmente la suma de las 9 casillas anteriores)

Un ejemplo de esta función, que creo que no necesita explicación, es el siguiente:

```
import "mod_key";
import "mod_effects";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_video";

global
  int png, png2,png3;
  int filter[]=8,20,3,4,5,6,7,8,9,70;
end

process main()
begin
  set_mode(640,480,16);

//Creo una imagen compuesta por tres cuadrados concéntricos
  png=new_map(50,50,16);map_clear(0,png,rgb(255,0,255));
  png2=new_map(30,30,16);map_clear(0,png2,rgb(255,255,0));
  png3=new_map(10,10,16);map_clear(0,png3,rgb(0,255,255));
  map_put(0,png2,png3,15,15);
  map_put(0,png,png2,25,25);


  bola(100,100,"Normal",0);
  bola(400,100,"Filtrado",1);
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

process bola(x,y,string txt,int f)
begin
  write(0,x,y-40,4,txt);
  size=200; graph=map_clone(0,png);
  if(f==1) filter(0,graph,&filter); end
  loop
    frame;
  end
end
```

Se pueden probar diferentes valores para la vector y observar los resultados.

Esta función, junto con **blur()**, se pueden utilizar para lograr un efecto de antialiasing en gráficos, fuentes, primitivas.... "Antialiasing" es el nombre que se le da al difuminado de los bordes de una figura, usado para evitar la visualización de feos "escalones". En el dibujo siguiente, al rombo de la derecha se le ha aplicado algún tipo de antialiasing y al de la izquierda no.



| **GRAYSCALE(LIBRERIA,GRAFICO,MODO)** |
|---|
| Esta función, definida en el módulo "mod_effects", cambia un gráfico a una escala de colores especificada por el tercer parámetro, el modo. Sólo funciona en modo de vídeo de 16 bits (y con imágenes de dicha profundidad).<br><br>El modo puede ser uno de los siguientes:<br>• 0: RGB -escala de grises-<br>• 1: R -escala de rojos-<br>• 2: G -escala de verdes- |

- 3: B -escala de azules-
- 4: RG
- 5: RB
- 6: BG

PARAMETROS:
INT LIBRERIA: Identificador de la librería cargada con **load_fpg()** ó 0 si no es el caso.
INT GRAFICO: Número de gráfico dentro de esa librería, o identificador de la imagen si se cargó independiente
BYTE MODO: Modo que define la escala de colores a usar

Un ejemplo de esta función que creo que no necesita explicación, es el siguiente:

```
import "mod_key";
import "mod_effects";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_video";

global
  int png, png2,png3;
  int filter[]=8,20,3,4,5,6,7,8,9,70;
end

process main()
begin
  set_mode(640,480,16);

//Creo una imagen compuesta por tres cuadrados concéntricos
  png=new_map(50,50,16);map_clear(0,png,rgb(255,0,255));
  png2=new_map(30,30,16);map_clear(0,png2,rgb(255,255,0));
  png3=new_map(10,10,16);map_clear(0,png3,rgb(0,255,255));
  map_put(0,png2,png3,15,15);
  map_put(0,png,png2,25,25);

  bola(10,50,"T 0",0);
  bola(110,100,"T 1",1);
  bola(210,150,"T 2",2);
  bola(310,200,"T 3",3);
  bola(410,250,"T 4",5);
  bola(510,300,"T 5",6);
  bola(610,350,"T 6",7);

  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

process bola(x,y,string txt,int modo)
begin
  write(0,x,y-20,4,txt);
  graph=map_clone(0,png);
  grayscale(0,graph,modo);
  loop
    frame;
  end
end
```

**RGBSCALE(LIBRERIA,GRAFICO,R,G,B)**

Esta función, definida en el módulo “mod_effects”, cambia un gráfico a una escala de colores especificada en forma de componentes RGB. El valor de R,G y B debe estar comprendido entre 0 y 1. Sólo funciona en modo de vídeo de 16 bits (y con imágenes de dicha profundidad).

PARAMETROS:

INT LIBRERIA: Identificador de la librería cargada con **load_fpg()** ó 0 si no es el caso.
INT GRAFICO: Número de gráfico dentro de esa librería, o identificador de la imagen si se cargó independiente
FLOAT R: Cantidad de rojo
FLOAT G: Cantidad de verde
FLOAT B: Cantidad de azul

Un ejemplo de uso de esta función podría ser el siguiente:

```
import "mod_key";
import "mod_effects";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_video";
import "mod_timers";
import "mod_rand";

global
  float r,g,b;
  int rr,rg,rb;
  float rgb[]=0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0;
  int png, png2, png3;
end

process main()
begin
  set_mode(320,240,16);
  x=160; y=120; size=200;

//Creo una imagen compuesta por tres cuadrados concéntricos
  png=new_map(50,50,16);map_clear(0,png,rgb(255,0,255));
  png2=new_map(30,30,16);map_clear(0,png2,rgb(255,255,0));
  png3=new_map(10,10,16);map_clear(0,png3,rgb(0,255,255));
  map_put(0,png2,png3,15,15);
  map_put(0,png,png2,25,25);
  graph=map_clone(0,png);

  write(0,160,70,4,"Valores R G B:");
  write_var(0,110,80,4,r);
  write_var(0,160,80,4,g);
  write_var(0,210,80,4,b);
  timer[0]=0;
  repeat
    if(timer[0]>100)
      unload_map(0,graph);
      graph=map_clone(0,png);
      rr=rand(0,10);
      rg=rand(0,10);
      rb=rand(0,10);
      r=rgb[rr];
      g=rgb[rg];
      b=rgb[rb];
      rgbscale(0,graph,r,g,b);
      timer[0]=0;
    end
    frame;
  until(key(_esc))
end
```

En el ejemplo anterior se muestra una imagen a la que cada segundo (más o menos) se le cambia su escala de colores de forma aleatoria. Para ello se utiliza un vector con diez valores posibles de las componentes R,G y B. Los que se aplican a la imagen de forma efectiva son seleccionados a partir estos valores utilizando aleatoriamente una posición del vector diferente cada segundo.

## *Trabajar con fuentes FNT

Algo muy básico en cualquier juego que se precie es mostrar textos: menús, el título, los puntos, el jugador al que le toca, y así un largo etcétera. Y mostrarlos con tipos de letra (las fuentes) atractivos y vistosos.

Bennu no incorpora ningún editor de fuentes, por lo que tendremos que recurrir, si queremos escribir textos espectaculares, a alguna aplicación externa. El problema es que el formato de las fuentes que utiliza Bennu, el formato FNT, es exclusivo de él y ningún otro programa lo utiliza. Este hecho proviene de la herencia que tiene Bennu de DIV. Por tanto, ya que este formato es tan específico y peculiar, ningún editor de fuentes "normal" que encontremos en el mercado podrá crear las fuentes con el formato usado por Bennu. Así que, ¿cómo creamos las fuentes FNT? Bueno, los usuarios de Windows tienen la suerte de disponer de una aplicación, "FNTEdit", que soluciona el problema. "FNTEdit" es un fenomenal editor de fuentes FNT, y se puede descargar desde la web de Bennu. Recuerda que también esta utilidad viene incluida dentro del BennuPack. Respecto los usuarios de Linux, éstos tienen dos alternativas: o bien utilizar Wine para hacer funcionar "FNTEdit" sobre su sistema, o bien programarse con comandos Bennu un editor de fuentes FNT a su medida, tal como se hizo de forma similar cuando se habló de contenedores FPG (ver capítulo 3).

Si se desea utilizar el "FNTEdit", el funcionamiento de esta aplicación es muy simple. A la hora de crear o modificar una fuente deberemos cargar de entrada alguna de las fuentes OpenType -archivos con extensión .otf- que tengamos preinstaladas en el sistema (éste es el formato de las fuentes que usamos por ejemplo en el Word y aplicaciones más comunes ya que es el formato "estándar", no como FNT), y a partir de ahí podremos definir su tamaño, el estilo, su color, su sombreado (y su color, y su orientación...), su borde (y su color), incluir imágenes,etc. Cuando ya hayamos acabado, crearemos la fuente dándole al botón "Generar" y entonces podremos ver en la parte inferior de la ventana una muestra de lo que sería un texto escrito con dicha fuente. Si estamos contentos, guardaremos el estilo de fuente en un archivo *.FNT. Recuerda que lo que grabamos es la fuente de letra en "abstracto", no es ningún texto concreto ni nada.

El formato estándar OpenType engloba los antiguos formatos TrueType y Type1 y es el más extendido en todos los sistemas operativos, usado en toda clase de aplicaciones, pudiendo crear documentos utilizando fuentes con dicho formato sin problemas de interoperabilidad. La gran diferencia entre estos formatos "estándar" y el formato FNT es similar a la que hay entre los gráficos vectoriales y los mapa de bits. Los tipos de letra tradicionales (OTF, TTF, etc) son a menudo tipos de letra escalables (es decir, en el mismo fichero se incluye información para dibujar textos a cualquier resolución y tamaño) y en blanco y negro, mientras los tipos de letra en Bennu no son escalables (no es posible dibujar un texto a varios tamaños diferentes con la misma fuente) pero a todo color. El formato FNT permite almacenar tipos de letra de formato "bitmap" (cada carácter dentro de la fuente es un gráfico compuesto de pixels), de 1, 8, 16 ó 32 bits por píxel en formato comprimido, y con píxels transparentes.

En Windows, para ver qué fuentes tienes instaladas en tu ordenador simplemente tienes que dirigirte a la carpeta "Fonts" que está dentro de la carpeta "Windows" que esté colgando directamente de la letra de la partición donde hayas instalado Windows (normalmente, C:). En sistemas GNU/Linux, las fuentes suelen estar guardadas dentro de la carpeta */usr/share/fonts*, suelen estar gestionadas por el servidor de fuentes Xft (parte integrante del servidor X) a través de la librería FreeType (la cual se encarga propiamente de renderizar las fuentes) y suelen ser localizadas y accedidas por los diferentes programas gracias a la suite de comandos *fontconfig*. No obstante, estos detalles no serán necesarios conocerlos ya que tanto en Windows como GNU/Linux nos ofrecen de serie software gráfico (los "gestores de fuentes") que nos permitirá visualizar y manejar con más comodidad y soltura las fuentes instaladas en nuestro sistema.

Una clasificación general que verás frecuentemente si profundizas en el mundo de la tipografía es distingir entre fuentes "Serif" y fuentes "Sans serif", independientemente del formato del archivo que tengan. Las fuentes "Sans serif" son las fuentes que no usan serifs, es decir, pequeñas líneas decorativas al final de los caracteres que embellecen su forma básica. Fuentes "Sans serif" populares son: Helvetica, Arial,Geneva,…Las fuentes "Sans serif" son más difíciles de leer, y por eso son usadas frecuentemente en textos cortos como encabezados de línea, títulos o mayúsculas. Las fuentes "Serif" son las fuentes que incorporan serifs,y las más populares son: Times Roman, Courier, New Century,etc.

| Fuente Serif | Fuente Sans Serif |
|---|---|

| | |
|---|---|
|  |  |

Si quieres conseguir fuentes directamente descargables de Internet,hay muchas web que las ofrecen gratuitamente. A continuación presento una lista, ni mucho menos exhaustiva, de sitios donde podremos conseguir gran cantidad de fuentes estándares OpenType (o en su defecto, TrueType), aunque no FNT, pero eso no importa porque sabemos que a partir de una estándar podemos crear una fuente FNT con "FNTEdit". Puedes completar tú mismo la lista si vas a cualquier buscador de Internet: aparecerán multitud de sitios más.

http://www.dafont.com
http://fonts.tom7.com
http://www.free-fonts.com
http://www.goodfonts.org
http://www.urbanfonts.com
http://www.fontspace.com
http://www.fontcubes.com
http://www.fontplus.net
http://www.fontreactor.com
http://www.pimpmyfonts.com
http://fawnt.com
http://www.searchfreefonts.com
http://www.identifont.com
http://www.openfontlibrary.org
http://thefreesite.com/Free_Fonts

Desde ésta última se puede acceder, entre otras, a:

http://www.1001freefonts.com
http://www.freetype.org

También puedes instalar y utilizar editores de fuentes "estándares" que te ofrezcan más posibilidades de diseño, para posteriormente utilizar el "FNTEdit" para pasarlas a formato FNT. Algunos programas editores de fuentes (TrueType y OpenType) que puedes utilizar podrían ser (la lista no es ni mucho menos completa):

**Fontforge** (http://fontforge.sourceforge.net). Esta aplicación es libre y multiplataforma
Un creador on-line: http://www.yourfonts.com
Un creador on-line: http://fontstruct.fontshop.com
**FontLab** (http://www.fontlab.com) Trial version
**FontCreator Program** (http://www.high-logic.com) Trial version

Por otro lado, es posible que te interese también utilizar algún visor de fuentes, como por ejemplo:

**FontMatrix** (http://fontmatrix.net) Esta aplicación es libre y multiplataforma
**Font Manager** (http://code.google.com/p/font-manager) Para GNU/Linux
**Fonty Python** (http://otherwise.relics.co.za/wiki/Software/FontyPython Para GNU/Linux
Un visor on-line: http://myfontbook.com

Bien, una vez que ya tenemos las fuentes creadas y listas, ¿cómo las utilizamos dentro de nuestro videojuego? Como no podía ser menos, Bennu incorpora una serie de funciones que posibilitan el uso en nuestros programas de fuentes para los textos que aparecen en la pantalla. Las fundamentales serían:

| LOAD_FNT("FICHERO") |
|---|
| Esta función, definida en el módulo "mod_map", carga en memoria un tipo de letra desde disco.,devolviendo un número entero que identifica al nuevo tipo de letra. Es posible utilizar a partir de entonces este valor con cualquier función que admita un tipo de letra como parámetro, como por ejemplo **write()**.<br><br>Nota: El formato FNT de 8 bits incluye también una paleta de colores, pero sólo puede haber una paleta de colores activa en todo momento, por lo que todo gráfico de 8 bits existente debe compartir la misma paleta de colores. Bennu usará automáticamente la paleta de colores del primer gráfico en recuperar de disco e ignorará la paleta de los demás. Esto quiere decir que si se cargan varios ficheros FNT de 8 bits con la función **load_fnt()** uno detrás de otro, a pesar de que cada fuente tenga su paleta propia, sólo se utilizará la paleta del primer fichero FNT, con lo que los colores de los siguientes tipos de fuente no serán los mismos que los originales (asociados a cada paleta particular). Es por eso que se recomienda, si se van a usar fuentes de 8 bits, utilizar una paleta común para todas ellas para evitar sorpresas. Otro truco sería crear las fuentes de 1 bit y utilizar luego la función **set_text_color()**.<br><br>PARÁMETROS: STRING FICHERO: Nombre del fichero<br><br>VALOR RETORNADO: INT : Identificador del nuevo tipo de letra |

| UNLOAD_FNT(IDFUENTE) |
|---|
| Esta función, definida en el módulo "mod_map", libera la memoria ocupada por un tipo de letra recuperado anteriormente de disco por la función **load_fnt()**<br><br>Se considera un error utilizar a partir de ese momento ese identificador de fuente en cualquier llamada a función, así como mantener en pantalla textos de esta fuente escritos con la función **write()** u otra equivalente.<br><br>PARÁMETROS: INT IDFUENTE : Identificador del tipo de letra |

Es decir, utilizando **load_fnt()** ya dispondremos de un identificador de fuente, al igual que con **load_png()** disponíamos de un identificador de Png. Y este identificador de fuente lo podremos utilizar como valor del primer parámetro de las funciones **write()** o **write_var()**, las cuales hasta ahora habíamos usado poniendo en dicho parámetro el valor de 0 -la fuente del sistema-. Y ya está: así de sencillo. Es decir, que si se supone que tenemos creada una fuente llamada "a.fnt", para utilizarla en un texto de nuestro programa, tendríamos que hacer lo siguiente:

```
import "mod_text";
import "mod_map";
import "mod_key";

process main()
private
    int idfuente;
end
begin
    idfuente=load_fnt("a.fnt");
    write(idfuente,320,240,4,"Hola");
    loop
        if(key(_esc)) break; end
        frame;
    end
    unload_fnt(idfuente);
end
```

A parte de las funciones de carga y descarga de fuentes, Bennu también dispone de unas cuantas funciones más relacionadas con la tipografía, que aunque no son tan comunes, al menos son interesantes de conocer, sobre todo porque ofrecen la posibilidad de crear nuestras propias fuentes FNT sin la necesidad de utilizar el "FNTEdit", tal como se comentó algunos párrafos más arriba. Estas funciones son las siguientes:

| NEW_FNT(PROFUNDIDAD) |
|---|
| Esta función, definida en el módulo "mod_map", crea un tipo de letra FNT en memoria. El nuevo tipo de letra estará preparado para contener caracteres de la profundidad de color deseada, pero en un principio estará vacío. Será |

necesario utilizar la función **set_glyph()** para añadir gráficos de caracteres al tipo de letra.

PARÁMETROS: INT PROFUNDIDAD : Profundidad de color (1,8,16 ó 32)

VALOR RETORNADO: INT : Identificador del nuevo tipo de letra

**GET_GLYPH(IDFUENTE,CARACTER)**

Esta función, definida en el módulo "mod_map", crea un gráfico en memoria (perteneciente a la librería 0) que contiene una copia de uno de los caracteres de un tipo de letra dado. De esta manera, se podrá modificar este gráfico en memoria según las necesidades de cada uno (sin alterar en absoluto la fuente original) y guardar luego si se desea los cambios en la propia fuente -o en otra-, empleando para ello la función **set_glyph().**

Si el carácter especificado en el segundo parámetro no existe dentro de la fuente especificada en el primero, esta función devolverá 0.

El gráfico creado es una copia fiel del contenido en el tipo de letra, y se incluyen dos puntos de control (el 1 y el 2) para almacenar información adicional del carácter:

- El punto 1 contendrá el desplazamiento a efectuar sobre el punto de escritura en el momento de escribir el carácter. Normalmente ocupará la esquina superior izquierda (0, 0) indicando que no hay ningún desplazamiento.
- El punto 2 contendrá el avance efectivo de posición, horizontal y vertical, que se realizará al escribir el carácter. El avance vertical no se emplea actualmente.

PARÁMETROS:

INT IDFUENTE : Identificador del tipo de letra
INT CARACTER: Número de carácter de la tabla ASCII (de 0 a 255)

VALOR RETORNADO: INT: Número de gráfico generado a partir del carácter de la fuente especificado

Un ejemplo de uso es el siguiente, donde a partir del gráfico correspondiente al carácter "Z" de la fuente del sistema, se genera otro gráfico (que asociamos a ***GRAPH***) al que seguidamente le modificamos su tamaño y orientación original para mostrar que su dibujo es diferente del dibujo del carácter "Z" en el cual se ha basado.

```
import "mod_map";
import "mod_key";
import "mod_video";
import "mod_string";

process main()
begin
   graph=get_glyph(0,asc("Z"));
   size=1000;
   x=160;
   y=120;
   flags=5;
   angle=90000;
   while(!key(_esc))
       frame;
   end
end
```

**SET_GLYPH (IDFUENTE,CARACTER,LIBRERIA,GRAFICO)**

Esta función, definida en el módulo "mod_map", permite cambiar cualquiera de los caracteres de un tipo de letra dado, por el gráfico especificado entre sus tercer y cuarto parámetros (que puede ser un gráfico obtenido previamente con **get_glyph()** y modificado posteriormente, o bien obtenido a partir de **new_map()**, **load_png()** o similares).

La acción de esta función cambia efectivamente el aspecto de la fuente en memoria, y el nuevo carácter será reflejado en posteriores usos de la fuente, incluyendo cualquier texto escrito por la función **write()** u otra equivalente, que

pudiera estar en pantalla en ese momento.

Es posible incluir dos puntos de control con información adicional sobre el carácter:

- El punto 1 contendrá el desplazamiento a efectuar sobre el punto de escritura en el momento de escribir el carácter. Normalmente ocupará la esquina superior izquierda (0, 0) indicando que no hay ningún desplazamiento.
- El punto 2 contendrá el avance efectivo de posición, horizontal y vertical, que se realizará al escribir el carácter. El avance vertical no se emplea actualmente.

Si estos puntos de control no se incluyen, la información del caracter existente no será modificada. Normalmente esto no es deseable, así que se recomienda añadir siempre los puntos de control al gráfico mediante la función **set_point()**.

PARÁMETROS:

INT IDFUENTE : Identificador del tipo de letra
INT CARACTER: Número de carácter de la tabla ASCII (de 0 a 255)
INT LIBRERIA: Identificador de librería FPG
INT GRAFICO: Número de gráfico dentro de la librería FPG

**SAVE_FNT(IDFUENTE,"FICHERO")**

Esta función, definida en el módulo "mod_map", crea o sobreescribe un fichero en disco, con el nombre especificado, que contenga un tipo de letra FNT actualmente ubicado en memoria. Este tipo de letra habrá sido recuperado de disco previamente por la función **load_fnt()** u otra equivalente. Cualquier cambio efectuado en memoria al tipo de letra con funciones como **set_glyph()** se guardará. Además, el fichero se guardará en formato comprimido.

PARÁMETROS:

INT IDFUENTE : Identificador del tipo de letra
STRING FICHERO : Nombre del fichero

Vamos a utilizar los comandos acabados de ver para hacer un primer ejercicio de creación de fuentes FNT propias (de 32 bits; si no se dice lo contrario, siempre trabajaremos en esa profundidad) a través de código Bennu, sin utilizar el "FNTEdit" u otras aplicaciones externas. Para empezar, haremos una prueba básica sin mucha utilidad práctica pero sencilla. Crearemos una fuente que contendrá sólo dos símbolos, y además éstos serán iguales: un cuadrado gris. Estos dos símbolos representa que se corresponderían con lo que sería los caracteres "A" y "B" de nuestra fuente. Es decir, haremos una fuente con la que sólo se puedan escribir la "A" y la "B", pero que en vez de escribir estos caracteres, se pintarán sendos cuadrados grises.

```
import "mod_map";
import "mod_key";
import "mod_draw";
import "mod_text";

process main()
private
    int font,map;
end
begin
//Genero un gráfico (un cuadrado gris)
map=new_map(10,10,32);
drawing_map(0,map);
drawing_color(rgb(100,100,100));
draw_box(0,0,9,9);
//Genero la fuente de 32 bits
font=new_fnt(32);
/*Este gráfico (el cuadrado gris) corresponderá al carácter 65 de la nueva fuente (El
carácter 65 de la tabla ASCII corresponde a la "A"). Como sólo se asigna este carácter,
el resto de letras y símbolos no serán visibles ya que no estarán definidos para esta
fuente: habría que hacer 255 set_glyph(), una por cada carácter de la nueva fuente
creada*/
set_glyph(font,65,0,map);
/*Sólo imprimirá un cuadrado gris correspondiente a "A".  "B" no se escribirá porque no
```

```
tiene asociado ningún símbolo de la fuente.*/
write(font,0,0,0, "A B");
while (!key(_SPACE)) frame; end //Fotograma y pausa
//Ahora hago que "B" también se escriba como el mismo cuadrado gris
set_glyph(font,66,0,map);
//Ahora se imprimen dos cuadrados grises
write(font,0,20,0, "A B");
/*Guardo la fuente creada (que está en memoria), en el disco, para poderla utilizar
posteriormente en otros programas como cualquier otra fuente.*/
save_fnt(font,"mifuente.fnt");
while(!key(_esc))frame; end
end
```

Fíjate en un detalle del código anterior. Mientras no pulsamos la tecla SPACE, aparece un sólo cuadrado gris, ya que sólo tenemos definido el caracter "A" en nuestra fuente, y por lo tanto "B" no se muestra. Pero después de pulsar SPACE, aparece como era esperable los dos cuadrados grises en el nuevo **write()**, pero además ¡también aparece el segundo cuadrado en el **write()** que habíamos escrito antes! Esto es porque los **write()** , aunque no se vea, se están actualizando constantemente y por lo tanto, en el momento que la fuente de letra utilizada es modificada, todos los textos impresos por esa fuente de letra se modifican de forma acorde, automáticamente.

Vamos ahora a crearnos una fuente un poco más útil, siguiendo la misma filosofía. Se escribirá primero un carácter utilizando la fuente del sistema; seguidamente, se modificará éste con **write_in_map()** y efectos propios de gráficos; finalmente, este carácter modificado se guardará en una nueva fuente y ésta se utilizará para volver a escribir el mismo carácter, comprobando así su nuevo aspecto.

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_string";
import "mod_effects";

process main()
private
          int font,graf;
end
begin
write(0,150,50,4,"Letra 'a' con la fuente del sistema : a");
/*Transformo el carácter pasado por parámetro a una imagen de tamaño dado por el segundo
parámetro y con un flag dado por el tercer parámetro (además, le aplico difuminado). El
valor devuelto es el código identificador de esta imagen*/
graf=char2grafchar("a",300,1);
font=new_fnt(32);
/*Hago que el caracter correspondiente al símbolo "a" de la fuente "font" sea
representado por la imagen acabada de generar en la línea anterior*/
set_glyph(font,asc("a"),0,graf);
//Muestro el resultado
write(0,150,100,4, "Letra 'a' con la fuente acabada de generar : ");
write(font,300,100,4, "a");
/*Y creo otro archivo en disco con esta nueva fuente (que consta únicamente de un
caracter por ahora), para poderla utilizar posteriormente en otros programas de forma
normal*/
save_fnt(font,"mifuente2.fnt");

while(!key(_esc)) frame; end
end


/*Función que transforma un carácter pasado por parámetro en una imagen, de un tamaño y
flags dados, y con difuminado*/
function char2grafchar(string letra,int nuevotam, int nuevoflag)
private
          int blend,graf,graf2;
          int alto,ancho;
end
begin
//Convierto el carácter pasado por parámetro en una imagen tal cual.
```

```
  graf=write_in_map(0,letra,4);

/*Las líneas siguientes sirven para cambiar el aspecto de la imagen generada por
write_in_map. No puedo utilizar variables como SIZE,FLAGS,etc porque estoy dentro de una
función y ahí éstas variables no tienen sentido (y estoy en una función y no un proceso
porque necesito devolver el identificador del gráfico modificado con return). Tampoco
existe una función específica que cambie las propiedades de un gráfico como su tamaño o
su flags si éste no pertenece a un proceso. Pero lo que sí puedo utilizar para modificar
el tamaño, flags,etc de un gráfico es función map_xput. Esta función "pega" en un gráfico
destino una copia de un gráfico origen, opcionalmente modificado en su tamaño y flags,
entre otras cosas. Así que lo que haremos será crear un gráfico destino vacío (llamado
graf2), donde "pegaremos" el gráfico graf y lo haremos cambiando su tamaño y su flag. De
esta manera, obtendremos en graf2 el gráfico modificado del carácter, y éste gráfico será
el que la función retornará*/

  ancho=map_info(0,graf,g_width);
  alto=map_info(0,graf,g_height);
/*Creamos el gráfico vacío cuyo tamaño será el que tiene el gráfico original (graf)
variando su tamaño según el que se ha indicado como segundo parámetro. Éste parámetro
está escrito en las mismas unidades que la variable SIZE (es decir, en tanto por ciento),
por lo que si se quiere modificar el tamaño en píxeles, habrá que dividir por 100. Es
decir, en este caso, si el 2º parámetro vale 300, el tamaño de graf2 será el triple, por
lo que tendremos que multiplicar el alto y el ancho del gráfico original por 3
(300/100).*/
  graf2=new_map(ancho*nuevotam/100,alto*nuevotam/100,32);
/*La línea clave: copiamos graf en graf2 en la coordenada central de éste (que se calcula
simplemente dividiendo por dos el ancho y alto de graf2 (ancho*nuevotam/100 y
alto*nuevotam/100 respectivamente), y ADEMÁS, lo copiamos con un nuevo tamaño y con un
nuevo flag. También podríamos haberlo copiado con un nuevo angle, pero no lo hemos
hecho...Después de esta línea ya tenemos en graf2 el gráfico modificado del caracter
original que queríamos*/
  map_xput(0,graf2,graf,ancho*nuevotam/200,alto*nuevotam/200,0,nuevotam,nuevoflag);

//Aplico también un difuminado al gráfico
  blur(0,graf2,3);

/*Las siguiente líneas comentadas sirven para aplicar a graf2 un efecto de tintado en
verde y de transparencia.Para ello utilizan las llamadas tablas blendop, tema que
requiere un poco más de estudio y que se explicará detalladamente en el último capítulo
de este manual. No obstante,si se desea observar el efecto que producen estas líneas, se
pueden descomentar y volver a ejecutarel programa*/

  /*blend=blendop_new();
  blendop_tint(blend,1.0,145,245,45);
  blendop_translucency(blend,0.4);
  blendop_assign(0,graf2,blend);*/

//Devuelvo el código del gráfico transformado
   return graf2;
end
```

No obstante, lo más normal no es hacer lo que hemos hecho en el ejemplo anterior (crear los nuevos caracteres de la fuente "al vuelo" con **write_in_map()**) sino que normalmente los caracteres se diseñan con un programa específico: un editor de fuentes (o incluso un editor de bitmaps o vectoriales) o bien mediante los servicios que ofrecen alguna de las webs listadas algunos párrafos más arriba. Y lo único que hacemos para generar la fuente FNT con nuestro programa escrito en Bennu es invocar a **set_glyph()** con cada uno de los diseños de esos caracteres previamente dibujados en el programa externo.

Comprobemos si ya hemos aprendido cómo funcionan las funciones de tratamiento de fuentes: sin probar el código siguiente, deberías ser capaz de saber qué hace.

```
import "mod_map";
import "mod_key";
import "mod_text";

process main()
private
```

```
      int idmifuente;
      int graf;
      int i;
End
BEGIN
idmifuente=fnt_new(32);
for(i=1;i<256;i++)
    graf=get_glyph(0,i);
    //...Modifico graf como quiera
    set_glyph(idmifuente,i,0,graf);
end
write(idmifuente,160,120,4,"Hola!");
loop
     frame;
     if(key(_esc)) break;end
end
save_fnt(idmifuente,"lala.fnt");
unload_fnt(idmifuente);
end
```

Una vez dominadas las órdenes de tratamiento de fuentes, podríamos empezar a desarrollar código útil. Por ejemplo, podemos crearnos nuestras propias funciones para convertir las imágenes incluidas en un FPG a carácteres de una fuente FNT, o a la inversa, transformar cada uno de los carácteres de una fuente FNT en una imagen incluida en un FPG. Como muestra, el siguiente código, que utiliza las funciones fpg2fnt() y fnt2fpg(); la primera de ellas tiene dos parámetros que representan el nombre (ó ruta) del FPG existente y el nombre (o ruta) del fichero FNT que generaremos, y la segunda tiene también dos parámetros que representan el nombre (ó ruta) del fichero FNT existente y el nombre (o ruta) del FPG que generaremos. En el código hemos supuesto que tenemos un FPG llamado “graficos.fpg” y una fuente FNT llamada “fuente.fnt”, cambia los nombres por los de algún fichero que tengas más a mano.

```
import "mod_map";
import "mod_key";
import "mod_text";

process main()
begin
 fpg2fnt("graficos.fpg","nueva_fuente.fnt");
 fnt2fpg("fuente.fnt","nuevo_fpg.fpg");
 while(!key(_esc))
    break;
 end
end

//A partir de un fpg preexistente crea una fuente con los 256 (0-255) gráficos de los
carácteres.
function fpg2fnt(string archivofpg, string archivofnt)
private
 int i;
 int nueva_fnt;
 int mifpg;
end
begin
 mifpg=load_fpg(archivofpg);
 nueva_fnt=new_fnt(32);
 from i= 0 to 255;
/*Cada imagen del fpg la convierto en un caracter de la nueva fuente. Hay que tener en
cuenta, no obstante, que los caracteres generados se colocarán en la misma posición que
tenían los gráficos en el fpg. Así pues, el gráfico 001 del fpg será el carácter 1 de la
fuente. Consulta la tabla ASCII para saber en qué posiciones dentro de la fuente conviene
generar los caracteres deseados, ya que por ejemplo, en el caso anterior, el carácter 1
no es imprimible.*/
      set_glyph(nueva_fnt,i,mifpg,i);
 end
 save_fnt(nueva_fnt,archivofnt);
end

//A partir de una fuente preexistente crea un fpg con cada uno de los gráficos de los
carácteres.
```

```
function fnt2fpg(string archivofnt, string archivofpg)
private
 int i;
 int miglip;
 int nuevo_fpg;
 int mifuente;
end
begin
 mifuente=load_fnt(archivofnt);
 nuevo_fpg=new_fpg();
 from i= 0 to 255;
/*Extraigo un caracter concreto de la fuente, lo añado al fpg y descargo ese caracter de
la memoria porqueno lo usaré más (en la próxima iteración iré al siguiente).*/
    miglip=get_glyph(mifuente,i);
    if(miglip!=0) //Siempre y cuando exista el caracter
       fpg_add(nuevo_fpg,i,0,miglip);
       unload_map(0,miglip);
    end
 end
 save_fpg(nuevo_fpg,archivofpg);
end
```

A partir de ejecutar el código anterior, podrás utilizar "nuevo_fpg.fpg" ó "nueva_fnt.fnt" en tus programas sin problemas (siempre que éstos estén en modo de vídeo de 32 bits, ya que los archivos generados son de esa profundidad).

## *Trabajar con regiones

Una "región de pantalla" no es más que una "ventana" invisible dentro de la ventana del juego, una zona invisible de la pantalla que puede contener una serie de gráficos, o no. De hecho, de forma predefinida, toda la ventana del juego es ya una región, la denominada región cero. Bennu nos permite definir hasta 31 regiones, de la 1 a la 31, del tamaño que nos de la gana, siempre que no se salgan de la zona de pantalla; la región cero no se puede definir por razones obvias: ya está definida.

Sus utilidades son muchas, por ejemplo, para saber si un gráfico está fuera o dentro de una zona de la pantalla, para mover el fondo, para definir ventanas para multijugador a pantalla partida,etc,pero la que más utilidad le vamos a dar es que podemos meter un gráfico dentro de una región, de tal manera que si éste se sale de ella, éste desaparecerá como si fuera el borde de la pantalla. Este efecto es muy socorrido en juegos en los que no toda la zona visible corresponde al área de juego (como por ejemplo se usan zonas de pantalla destinadas al uso de marcadores).

A modo de resumen, en conclusión, podemos ver que las regiones se pueden usar para:

*Definir el área ocupada por una región de scroll (en seguida veremos qué son y cómo funcionan).
*Definir un área de "clipping", de manera que los procesos que estén marcados como pertenecientes a la misma sean recortados por sus límites. Y en relación a esto, definir que si un gráfico sale de una región, desaparezca como si fuera el borde de pantalla, o viceversa
*Definir una zona de pantalla donde nos interesará saber fácilmente cuándo un proceso entra o sale.
*Definir ventanas multijugador a pantalla partida.
*Etc

Para poder trabajar con zonas que no sean la 0, necesitaremos crearlas antes. Y para ello utilizaremos la función **define_region()**

| **DEFINE_REGION(REGION,X,Y,ANCHO,ALTO)** |
|---|
| Esta función, definida en el módulo "mod_screen", permite definir los límites de una región. Existen 32 regiones disponibles, de las cuales la región 0 corresponde siempre a la pantalla completa y no puede cambiarse.<br><br>Las coordenadas se dan siempre en pixels, donde (0, 0) corresponde a la esquina superior izquierda de pantalla. Sin |

embargo, por sí mismo definir una región no tiene ningún efecto hasta que la región se utiliza.

Para ello, pronto veremos que puedes asignar el número de región a la variable ***REGION*** de un proceso, usarla con la función **start_scroll()**, o emplear también la función **out_region()** con la nueva región definida.

PARÁMETROS:

INT REGION: Número de región, de 1 a 31
INT X : Coordenada horizontal de la esquina superior izquierda de la región
INT Y : Coordenada vertical de la esquina superior izquierda de la región
INT ANCHO : Ancho en píxeles de la región
INT ALTO : Alto en píxeles de la región

Tal como acabamos de decir en el cuadro anterior, definir una región no tiene ningún efecto aparente sobre nuestro juego. Falta un paso más: utilizarla. Y hay varias maneras para eso. En el siguiente apartado ya hablaremos de los scrolls y de la aplicación que tienen las regiones a éstos, pero si no trabajamos con scrolls, el uso de las regiones suele ser el siguiente: normalmente se usan para establecer que uno o más procesos determinados sólo serán visibles dentro de una región en concreto. Y para lograr esto, lo que hay que hacer es asignar a la variable local ***REGION*** de ese/esos proceso/s el número de región definido previamente con **define_region()** donde queremos que el proceso sea visible.Las partes del gráfico del proceso que queden fuera de la zona definida por la región se recortarán y no serán mostradas. El valor por defecto de ***REGION*** para todos los procesos es 0, que equivale a la pantalla completa.

A continuación veremos un ejemplo donde se puede comprobar el efecto del uso de la variable ***REGION***. El código consiste en la modificación mediante los cursores del teclado tanto del origen de la esquina superior izquierda como de la altura y anchura de una determinada región numerada como 1.La manera de visualizar este cambio es asociar dicha región 1 en este caso al código principal del programa -podría haber sido un proceso cualquiera-. Como acabamos de decir que al asociar un proceso a una región, estamos haciendo que el gráfico de ese proceso sólo sea visible dentro de esa región, si modificamos la posición y tamaño de la región, modificaremos la porción de gráfico visible en cada momento, que es lo que tendría que ocurrir si lo pruebas.

```
import "mod_key";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_screen";

process main()
private
  int xr=0,yr=0; //Esquina superior izquierda de la región
  int wr,hr;  // Anchura y altura de la región
end
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
  x=160;
  y=120;
  region=1; //Sólo se verá el gráfico del código dentro de la región 1
  write(0,10,50,3,"LEFT : Disminuye anchura región");
  write(0,10,60,3,"RIGHT: Aumenta anchura región");
  write(0,10,70,3,"UP   : Disminuye altura región");
  write(0,10,80,3,"RIGHT: Aumenta altura región");
  write(0,10,100,3,"Anchura región: ");
  write_var(0,120,100,3,wr);
  write(0,10,110,3,"Altura región: ");
  write_var(0,120,110,3,hr);
  write(0,10,120,3,"X región: ");
  write_var(0,120,120,3,xr);
  write(0,10,130,3,"Y región: ");
  write_var(0,120,130,3,yr);
  repeat
    if(key(_left)  && xr>0)   xr=xr-2; end
    if(key(_right) && xr<160) xr=xr+2; end
    if(key(_up)   && yr>0)   yr=yr-2; end
    if(key(_down) && yr<120) yr=yr+2; end
/*Modifico la anchura y altura de la región acorde a la posición de su esquina superior
izquierda. Esta fórmula se podría cambiar por otra */
    wr=320-(xr*2);
```

```
    hr=240-(yr*2);
    define_region(1,xr,yr,wr,hr); //Defino la región 1
    frame;
  until(key(_esc))
end
```

Otra función importante respecto el tema de las regiones es **out_region()**.

| **OUT_REGION(ID,REGION)** |
|---|
| Esta función, definida en el módulo "mod_screen", devuelve 1 si un proceso cuyo identificador es su primer parámetro está fuera de la región indicada como segundo parámetro. En otras palabras, permite comprobar si un proceso ha sido dibujado dentro o fuera de una región en pantalla.<br><br>La función devuelve 1 si el proceso se encuentra completamente fuera de los límites de la región, y 0 si alguna parte o todo el proceso toca dentro de la misma.<br><br>Antes de llamar a esta función, es preciso haber establecido los límites de la región mediante **define_region()**, a no ser que se utilice la región 0, que corresponde siempre a toda la pantalla y ofrece una forma sencilla de comprobar si un proceso ha salido fuera de la pantalla.<br><br>Si quieres comprobar si el proceso actual ha salido fuera de una región, recuerda que se puede utilizar la variable local ***ID*** para obtener su identificador.<br><br>PARÁMETROS:<br><br>INT ID: Identificador de un proceso<br>INT REGION: Número de región, de 1 a 31 |

Un ejemplo de la función anterior podría ser el siguiente. En él tenemos cuatro procesos, cuyos identificadores se guardan cada uno en un elemento de la matriz *idb[]*. Además, tenemos definida una región (región 1), visualizada gracias a un cuadrado de diferente color. Los gráficos de dos de los procesos estarán entrando y saliendo continuamente de dicha región, y los gráficos de los otros dos procesos estarán entrando y saliendo continuamente de la pantalla (región 0).El programa muestra cuatro textos que se actualizan constantemente y muestran el estado de cada uno de los cuatro procesos respecto si están fuera o dentro de la región 1 (dos procesos) o de la región 0 (los otros dos). Y esto se sabe gracias a lo que devuelve **out_region()**. El estado -dentro o fuera- de cada elemento de *idb[]* se almacena en un elemento de la matriz *out[]*.

```
import "mod_key";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_screen";

process main()
private
  int idb[3];
  int out[3];
end
begin
/*Pinto un rectángulo visible con las mismas dimensiones y posición que la región 1 que
creo a continuación, para hacer ésta visualizable.*/
  graph=new_map(120,100,32);map_clear(0,graph,rgb(80,20,20));
  x=220;y=75;z=1;
/*Creo la región de forma real. Notar que el ancho y alto es igual al del rectángulo
pintado en la línea anterior, y la posición de su esquina superior izquierda (160,25)
viene dada por un simple cálculo a partir del centro del rectángulo pintado (220,75) y su
tamaño (120,100).*/
  define_region(1,160,25,120,100);

  write(0,10,60,3,"Out Region 0 Bola 1:");
  write_var(0,147,60,3,out[0]);
  write(0,10,70,3,"Out Region 0 Bola 2: ");
  write_var(0,147,70,3,out[1]);
  write(0,10,80,3,"Out Region 1 Bola 3: ");
  write_var(0,147,80,3,out[2]);
```

```
  write(0,10,90,3,"Out Region 1 Bola 4: ");
  write_var(0,147,90,3,out[3]);

  idb[0]=bola(300,160,100,0,1,1);
  idb[1]=bola(300,160,100,0,2,1);
  idb[2]=bola(270,105,50,1,3,1);
  idb[3]=bola(270,105,50,1,4,1);

/*Out[0] valdrà 1 si el proceso idb[0] sale fuera de la pantalla.
Out[1] valdrà 1 si el proceso idb[1] sale fuera de la pantalla.
Out[2] valdrà 1 si el proceso idb[2] sale fuera de la región 1.
Out[3] valdrà 1 si el proceso idb[3] sale fuera de la región 1.*/
  repeat
    out[0]=out_region(idb[0],0);
    out[1]=out_region(idb[1],0);
    out[2]=out_region(idb[2],1);
    out[3]=out_region(idb[3],1);
    frame;
  until(key(_esc))
  let_me_alone();
end

/*Al pasar por parámetro la variable local region, estamos indicando a la hora de crear
el proceso, en qué región queremos que éste sea visible*/
process Bola(x,y,size,region,t,m)
begin
  graph=new_map(8,8,32);map_clear(0,graph,rgb(100,100,100));
  loop
/*Las variables "t" y "m" simplemente son para controlar el movimiento de los diferentes
procesos. No tienen mayor importancia.*/
    switch(t)
      case 1:
        if(x>-50 && m==1) x=x-5; else m=2; end
        if(x<370 && m==2) x=x+5; else m=1; end
      end
      case 2:
        if(y>-50 && m==1) y=y-5; else m=2; end
        if(y<250 && m==2) y=y+5; else m=1; end
      end
      case 3:
        if(x>130 && m==1) x=x-2; else m=2; end
        if(x<310 && m==2) x=x+2; else m=1; end
      end
      case 4:
        if(y>0   && m==1) y=y-2; else m=2; end
        if(y<150 && m==2) y=y+2; else m=1; end
      end
    end
    frame;
  end
end
```

## *Trabajar con scrolls

Un scroll es un fondo más grande que la pantalla, y que se mueve. Los scrolls permiten mostrar en pantalla una zona de juego más grande que ésta, de manera que sólo es posible ver una parte del total.Podemos usar los “scrolls” para muchas cosas: para dibujar por ejemplo un paisaje de fondo que sigue a nuestro héroe en un juego de plataformas, para hacer un mundo que se mueve bajo los pies de nuestro protagonista en un juego de rol, para hacer planetas y nieblas que van apareciendo en el universo cuando nuestra nave lo va surcando de punta a punta....

Bennu nos permite usar hasta 10 scrolls a la vez. Esto puede ser útil para mostrar dos zonas de scroll distintas, pero el uso más común es mostrar dos ventanas de movimiento dentro de la misma zona (por ejemplo para partidas a dos jugadores). Cada uno de estos scrolls dispone además de no uno, sino dos fondos a distinta profundidad

que podremos mover a distintas velocidades. El fondo de segundo plano se verá a través de los píxels transparentes del gráfico del fondo de primer plano.

Podemos clasificar los scrolls en dos tipos básicos: los scrolls automáticos, y los scrolls que persiguen a un proceso. Antes de comenzar a codificar debemos tener claro qué tipo de scroll queremos implementar. Un scroll automático es aquél que, independientemente de los movimientos o posiciones de los procesos en pantalla, va a ir desplazando el fondo de manera automática en una o unas direcciones determinadas, a modo de fondo siempre animado. Este movimiento de scroll lo tendremos que programar "a mano" para hacer que se mueva de esta forma "mecánica" tal como queramos. El otro tipo de scroll, el que persigue a un proceso, consiste básicamente en hacer que el fondo se desplace junto con el movimiento de un determinado proceso: así, si el proceso está quieto, el fondo está quieto, y cuando el proceso se mueve en una dirección, el scroll se moverá en dicha dirección. Éste es el típico efecto que se utiliza en los juegos de rol a vista de pájaro (también llamados RPG, veremos un tutorial en el próximo capítulo) cuando se quiere mostrar el desplazamiento de nuestro protagonista a través de un territorio.

Para iniciar un scroll (del tipo que sea) hay que usar la función **start_scroll()**, (definida en el módulo "mod_scroll") en sustitución de **put_screen()**, (si es que estábamos usándola hasta ahora). Es decir, **start_scroll()** hace básicamente lo mismo que **put_screen()**: pinta una imagen de fondo (bueno, en realidad dos imágenes para dos fondos a distinta profundidad, ahora lo veremos), con la gran diferencia que esta/s imagen/es de fondo son capaces de moverse. **Put_screen()** pinta un fondo fijo, **start_scroll()** pinta un fondo móvil.

**START_SCROLL(NUMERO,FICHERO,GRAFICO,FONDO,REGION,FLAGS)**

Esta función, definida en el módulo "mod_scroll", inicia y pinta un scroll (que puede ser doble) con las características especificadas en sus parámetros. Con esta función se podrán iniciar hasta 10 scrolls (dobles o no) diferentes.

PARAMETROS:

INT NUMERO : Número identificador de scroll (0 a 9)
INT FICHERO : Número de fichero FPG que contiene los gráficos GRAFICO y FONDO (o 0 si los gráficos de scroll se han cargado individualmente, por ejemplo con **load_png(), new_map()** ó similar)
INT GRAFICO : Número de gráfico para el primer plano del scroll.Es obligatorio especificar un gráfico válido.
INT FONDO : Número de gráfico para el fondo del scroll, 0 para ninguno.
INT REGION: Número de región donde el scroll funcionará.Esto es útil si divides la pantalla y quieres que cada parte sea un scroll para un jugador distinto, o para hacer un pequeño mapa scrollable a modo de zona pequeña dentro de la ventana principal,por ejemplo. Si vas a usar toda la pantalla, el número que tienes que indicar es el cero.
INT FLAGS : Si este parámetro vale 0, el scroll se moverá hasta que el margen del gráfico del primer plano del fondo –el 2º parámetro- coincida con uno de los límites de la región/pantalla, momento en el que el scroll se parará, ya que se ha llegado al final de la imagen. Hay que tener en cuenta que este valor de 0 en este parámetro sólo tiene sentido en el caso de los scrolls "perseguidores" de procesos, porque con los scrolls automáticos precisamente lo que interesa es evitar esto, es decir, lo que interesa es conseguir que el scroll se repita a modo de mosaico indefinidamente de forma cíclica. Para ello, este parámetro puede además valer cualquiera de los siguientes valores:

| | |
|---|---|
| **1** | El gráfico de primer plano se repite horizontalmente. |
| **2** | El gráfico de primer plano se repite verticalmente. |
| **3 (2+1)** | El gráfico de primer plano se repite tanto horizontal como verticalmente. |
| **4** | El gráfico de fondo se repite horizontalmente. |
| **5 (4+1)** | El gráfico de fondo se repite horizontalmente y el de primer plano también. |
| **6 (4+2)** | El gráfico de fondo se repite horizontalmente y el de primer plano verticalmente |
| **7 (4+2+1)** | El gráfico de fondo se repite horizontalmente y el de primer plano en ambas direcciones |
| **8** | El gráfico de fondo se repite verticalmente. |
| **9 (8+1)** | El gráfico de fondo se repite verticalmente y el de primer plano horizontalmente |
| **10 (8+2)** | El gráfico de fondo se repite verticalmente y el de primer plano también. |
| **11 (8+2+1)** | El gráfico de fondo se repite verticalmente y el de primer plano en ambas direcciones |
| **12 (8+4)** | El gráfico de fondo se repite tanto horizontal como verticalmente. |
| **13 (8+4+1)** | El gráfico de fondo se repite en ambas direcciones y el de primer plano horizontalmente. |

| | |
|---|---|
| **14 (8+4+2)** | El gráfico de fondo se repite en ambas direcciones y el de primer plano verticalmente. |
| **15 (8+4+2+1)** | El gráfico de fondo se repite en ambas direcciones y el de primer plano también. |

Una vez iniciado el scroll, no hay que olvidarse de cerrarlo antes de acabar la ejecución del programa para que ésta se realice limpiamente, usando **stop_scroll()** pasándole el identificador de scroll que queremos parar

**STOP_SCROLL (NUMERO)**

Esta función, definida en el módulo "mod_scroll", detiene un scroll activado previamente mediante la función **start_scroll()** . Debe indicarse el mismo identificador de scroll especificado al inicializarla. Ese scroll no será visible a partir del próximo frame (como si hubiéramos hecho un **clear_screen()** para un **put_screen()**).

PARAMETROS:
INT NUMERO : Número identificador de scroll (0 a 9)

Vamos a ir conociendo cómo funciona el mecanismo de scroll a partir de un ejemplo que iremos ampliando a medida que vayamos aprendiendo más aspectos del mismo.

Para este ejemplo y todos los demás de este apartado, necesitarás tener dos imágenes que serán el fondo de primer plano y de segundo plano de nuestro scroll. Como el ejemplo lo haremos en una resolución de 640x480, estas imágenes han de ser más grandes que estas dimensiones para que se note algo; por ejemplo, de 1000x1000. Además, dichas imágenes tendrán que ser algo variadas en su dibujo para que se note más el scroll, e importante, la imagen que será la de primer plano ha de tener bastantes zonas transparentes (es decir, con sus componentes RGBA iguales a 0,0,0,0) para que se puedan ver en esas zonas las porciones de la imagen de segundo plano que coincidan. Estas dos imágenes las incluiremos en un FPG llamado "scroll.fpg": el primer plano con código 001 y el segundo con 002.

Además, en nuestro ejemplo crearemos un proceso que se podrá mover en las cuatro direcciones mediante el cursor. Su gráfico lo generaremos en memoria.

Empezamos con el siguiente código inicial:

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";

process main()
private
          int idfpg;
end
begin
set_mode(640,480,32);
idfpg=load_fpg("scroll.fpg");
start_scroll(0,idfpg,1,2,0,0);
proceso1();
loop
          if(key(_q)) stop_scroll(0); end
          if(key(_esc)) exit();end
          frame;
end
end

process proceso1()
begin
          graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
          x=320;y=240;
          loop
                    if(key(_up)) y=y-5;end
                    if(key(_down))y=y+5;end
                    if(key(_left))x=x-5;end
```

```
                        if(key(_right))x=x+5;end
                        frame;
                end
end
```

Lo que hacemos aquí es poner un fondo de pantalla mediante **start_scroll()**, y el proceso que se moverá encima de él. Fíjate que en las zonas transparentes del fondo de primer plano se tiene que ver lo que asoma del fondo de segundo plano. Los valores de **start_scroll()** son claros: creamos el scroll nº 0, a partir del FPG "scroll.fpg", el gráfico de primer plano será el primero dentro del FPG (código 001) y el de segundo plano será el segundo (´codigo 002); la región donde será visible el scroll es la región 0 (toda la pantalla) y no habrá mosaico de scroll (es decir, cuando los procesos se desplacen hasta el extremo de la imagen que forma el scroll no podrán ir más allá). Si en vez de cargar las dos imágenes del scroll a partir de un FPG, quisiéramos cargarlas individualmente con **load_png()**, podríamos haber escrito algo así como:*start_scroll(0,0,idpng1,idpng2,0,0);*

Fíjate también que si apretamos la tecla "q", desaparece el scroll, gracias a **stop_scroll()**. Normalmente, esta orden se escribe justo antes de acabar el programa, pero la hemos puesto aquí para demostrar su acción.

No obstante, tenemos un problema: de entrada el fondo no se mueve, ni siquiera cuando movemos el proceso fuera de los límites de la pantalla. El fondo está tan quieto como lo estaba con **put_screen()**. ¿Qué nos falta? Decidir si vamos a hacer un scroll automático o "perseguidor" de un proceso. Para empezar, en nuestro ejemplo vamos a implementar el scroll de tipo automático (luego ya veremos el otro). Para ello, antes has de saber lo siguiente.

Al igual que pasaba con el dispositivo ratón, una vez creado el scroll mediante **start_scroll()**, es posible ajustar diversos aspectos del mismo, incluso a cada frame, modificando una estructura global predefinida: en este caso la estructura llamada ***SCROLL***. De esta manera, no es necesario usar un proceso para trabajar con scrolls, aunque si recomendable por el orden. Más exactamente, como podemos definir 10 zonas de scroll, en realidad, de lo que disponemos es de una tabla de estructuras, donde ***SCROLL[0]*** será la estructura que contenga los datos de configuración para el scroll de nº 0, ***SCROLL[1]*** será la estructura correspondiente al scroll nº1, etc.

En concreto, para mover el scroll, ¿qué hay que hacer, pues?. La estructura ***SCROLL[n]*** tiene muchos campos, pero para mover el scroll vamos a usar los campos **"x0", "y0", "x1"** e **"y1"**. **"x0"** controla la coordenada x del primer plano del scroll y **"x1"** controla la coordenada x del segundo plano; de igual manera, **"y0"** controla la coordenada y del primer plano del scroll y **"y1"** la coordenada y del segundo plano, respecto la pantalla. Sólamente hay que establecer el valor que queremos a estos campos para desplazar el scroll a ese nuevo valor, igual que haríamos con las variables X e Y de cualquier gráfico de proceso.

Hay otro campo de la estructura ***SCROLL*** que puede que te interese, y es la variable **"ratio"** (usada así, por ejemplo: *scroll[0].ratio*). En ella se guarda la velocidad del segundo plano del fondo respecto al primer plano en porcentaje; así, si vale 100, el fondo se moverá a la misma velocidad que el plano principal, si vale 50, el fondo irá a la mitad de la velocidad y si vale 200 irá el doble de rápido (y si vale 33 irá a un tercio, etc.).

Vamos ya, pues, a implementar este tipo de scroll en el ejemplo anterior. Para ello, modificaremos el código añadiendo la siguiente línea dentro del bucle Loop/End del programa principal, justo antes de la orden frame;:

```
scroll[0].y0=scroll[0].y0 - 10;
```

Lo que hace esta línea, en teoría, es modificar la posición del gráfico del primer plano del scroll de manera que en cada frame se mueva 10 píxeles para arriba automáticamente. Pero si lo volvemos a probar, no pasa nada de nada...porque se nos ha olvidado cambiar el valor del último parámetro de **start_scroll()**, el parámetro de flags. Recordemos que si vale 0 no se va a poder ver ningún mosaico de imágenes de scroll, que es lo que necesitamos para crear un scroll de tipo automático, así que tendremos que cambiar ese valor. ¿Por cuál? Ya que con la línea que acabamos de añadir lo que estamos haciendo es mover el gráfico de primer plano del fondo verticalmente (de abajo a arriba), nos interesará generar un mosaico vertical para ese gráfico, así que valores apropiados podrían ser el 2,3,6,7,10,11,14 o 15. Si probamos con el 2 mismo, veremos que, efectivamente, ahora el gráfico de primer plano se mueve (y el de segundo plano se queda quieto).

Y de hecho, ocurrirá el mismo efecto pongamos el número que pongamos de la lista anterior como último parámetro del **start_scroll()**, ya que, si por ejemplo pones ahora allí un 10, puedes comprobar que el gráfico de segundo plano no se moverá, ya que este número sólo indica que ese gráfico "puede" crear un mosaico vertical, pero para ello se ha de mover en esa dirección, y eso lo tenemos que decir explícitamente añadiendo una línea más dentro del bucle Loop/End del programa principal, justo antes de la orden frame. Por ejemplo:

```
scroll[0].y1=scroll[0].y1 - 5;
```

Verás que haciendo esto, ahora los dos gráficos se mueven de abajo a arriba, aunque el de más al fondo lo hace la mitad de rápido. Podrías haber obtenido exactamente el mismo efecto que el que acabamos de conseguir si en vez de escribir la línea anterior, escribes esto:

```
scroll[0].ratio=200;
```

Esta línea lo que hace es mover el gráfico de segundo plano en la misma dirección en que lo haga el de primer plano, (sea cual sea), a una velocidad relativa a éste dada por el porcentaje escrito: en este caso, "200" quiere decir el doble de lento,(50 querría decir el doble de rápido, y así). Compruébalo.

Finalmente, y si pruebas ahora de poner un 3 como último parámetro de **start_scroll()**, ¿qué pasa?. Primero volvemos a dejar sin movimiento el gráfico de segundo plano porque aunque ahora tengamos escrita una línea que mueve dicho gráfico -bien usando "y1", bien usando "ratio"- , éste no tiene ya la posibilidad de hacer mosaico (es como si hubiéramos escrito un "0" de valor para ese fondo en **start_scroll()**). Y además, con el valor "3", estamos concediendo la posibilidad de que se produzca un mosaico horizontal, pero para verlo necesitaremos mover horizontalmente el gráfico de primer plano, añadiendo esta línea:

```
scroll[0].x0=scroll[0].x0 - 20;
```

Si escribimos esta nueva línea, verás que el scroll se mueve en diagonal hacia arriba a la izquierda: esto es evidentemente porque estamos moviéndolo arriba con "y0" y a la izquierda con "x0", y los efectos se suman.

Te recomendaría que probaras diferentes valores para el último parámetro de **start_scroll()** y comprobaras su efecto en el movimiento del fondo: es la mejor manera de aprender. Un dato curioso respecto al valor de dicho último parámetro: si escribes un valor que posibilite el mosaico horizontal y/o vertical de alguno de sus dos gráficos, y ese gráfico tiene unas dimensiones menores que el ancho y alto de la pantalla, lo que observarás es que con sólo escribir **start_scroll()**, se produce un efecto de mosaico de forma automática en el fondo de nuestro juego, sin tener que programarlo "a mano" ,como cuando escribimos el ejemplo expuesto en la explicación de la función **put()**.

Ya tenemos creado nuestro scroll "automático". Ahora vamos a por el otro tipo, el scroll "perseguidor". Volvemos a partir del código inicial que teníamos:

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";

process main()
private
          int idfpg;
end
begin
set_mode(640,480,32);
idfpg=load_fpg("scroll.fpg");
start_scroll(0,idfpg,1,2,0,0);
proceso1();
loop
          if(key(_q)) stop_scroll(0); end
          if(key(_esc)) exit();end
          frame;
end
end

process proceso1()
begin
          graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
          x=320;y=240;
          loop
                    if(key(_up)) y=y-5;end
                    if(key(_down))y=y+5;end
```

```
                    if(key(_left))x=x-5;end
                    if(key(_right))x=x+5;end
                    frame;
          end
end
```

y añadiremos otro proceso más, para notar mejor los efectos de este scroll (acuérdate de llamarlo desde el programa principal, y de incluir el módulo "mod_grproc", debido a la función **advance()**):

```
process proceso2()
begin
          graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
          x=220;y=200;
          loop
                    if(key(_a))angle=angle+5000;end
                    if(key(_d))angle=angle-5000;end
                    if(key(_s))advance(5);end
                    frame;
          end
end
```

Lo primero que hay que hacer es decir cuál va a ser el proceso que será perseguido por el scroll. Es decir, cuál será el proceso "cámara". Para ello, hay que asignar al campo **"camera"** del scroll correspondiente el identificador del proceso que queremos que persiga. Esto se puede hacer desde el código principal, pero lo más habitual es hacerlo al principio del código del propio proceso, así:

```
scroll[0].camera=id;
```

donde ***ID*** ya sabemos que es una variable predefinida local que devuelve el identificador de proceso del proceso actual. Prueba de añadir esta línea justo después del BEGIN de "proceso1()", por ejemplo, y prueba de mover este proceso con los cursores.

Parece que va, pero no del todo bien. Por ejemplo, si el proceso sale de los límites de la pantalla, desaparece, y lo chulo sería que se fuera viendo cómo éste se va desplazando por el scroll sin salirse nunca de la pantalla. Otra cosa que no queda bien es que "proceso2()" no participa del movimiento del scroll: se queda fijo, con lo que no es realista que nos desplacemos: los procesos también deberían de moverse al igual que el scroll para dar una sensación de distancia y posicionamiento fijo real dentro de un mapa scrolleable. ¿Cómo solucionamos estos dos problemas?

Bueno, en realidad estos dos efectos son consecuencia de un sólo problema. Por defecto, los procesos no son dibujados en ningún scroll,por lo que no responden a la lógica de éstos; es como si el scroll fuera por un lado, y los procesos por otro, sin enterarse de la existencia de los primeros. Hay que decir explícitamente que tal proceso o tal otro queremos que forme parte del scroll y así se comportará como esperamos. Y esto se hace con la variable local predefinida ***CTYPE*** ("Camera TYPE"). Si queremos que un proceso forme parte de un scroll y se mueva acorde con él, es preciso que la variable local del proceso ***CTYPE*** sea igual a la constante predefinida C_SCROLL (con valor real 1).

Importante: hecho este cambio, las variables de posición X e Y del proceso se considerarán relativas a la esquina superior izquierda del gráfico de primer plano del scroll y no a la de la pantalla.

Si queremos que un proceso deje de formar parte de un scroll para volver a su estado por defecto, ***CTYPE*** ha de ser igual a la constante predefinida C_SCREEN (con valor real 0).

Otra cosa importante: por defecto, los procesos con CTYPE=1 serán visibles en todos los scrolls existentes en nuestro juego. Para escoger un scroll concreto (o más) en los que un proceso es visible (esto nos podría interesar por ejemplo si queremos dividir la pantalla del juego para dos jugadores), hay que asignar a su variable local predefinida ***CNUMBER*** la suma de una o más de las constantes predefinidas C_0, C_1... hasta C_9, para indicar qué números de scrolls son aquellos en los que el proceso será visible. Así, con la línea *CNUMBER = C_0 + C_2* se hace que un proceso sea visible en las zonas de scroll 0 y 2, pero no en la 1 ni en ninguna otra (si existe). Los valores reales de estas constantes predefinidas, por si interesa, son:

| | |
|---|---|
| C_0 | 1 |

| | |
|---|---|
| C_1 | 2 |
| C_2 | 4 |
| C_3 | 8 |
| C_4 | 16 |
| C_5 | 32 |
| C_6 | 64 |
| C_7 | 128 |
| C_8 | 256 |
| C_9 | 512 |

El valor por defecto de ***CNUMBER*** es 0, que indica, como hemos dicho, que los procesos son visibles en todos los scrolls existentes. Esta variable sólo tiene sentido si se inicializa también la variable local ***CTYPE***.

Una vez que ya conocemos la existencia de estas dos nuevas variables predefinidas, vamos a usarlas para lo que nos proponemos. Si queremos que un proceso sea visible y pertenezca a nuestro scroll, lo que tendremos que escribir, al principio del código de ese proceso, las líneas:

```
ctype=c_scroll;
cnumber=0;
```

aunque ésta última no es necesaria porque su valor por defecto ya es el hemos puesto: si sólo quisiéramos que los procesos se vieran en el scroll 0 (que de hecho, es el único que hay), también podríamos haber puesto cnumber=c_0;. Puedes probar otros valores para ver lo que pasa.

Así pues, si has escrito ambas líneas en “proceso1()” y “proceso2()”, ya tendremos nuestro scroll “perseguidor” perfectamente funcional. Y ya está.

Si has seguido los pasos que se han ido marcando, al final tendrás que tener un código como éste:

```
import "mod_map";
import "mod_proc";
import "mod_grproc";
import "mod_text";

process main()
private
          int idfpg;
end
begin
set_mode(640,480,32);
idfpg=load_fpg("scroll.fpg");
start_scroll(0,idfpg,1,2,0,3);
proceso1();
proceso2();
loop
          if(key(_q)) stop_scroll(0); end
          if(key(_esc)) exit();end
          scroll[0].y0=scroll[0].y0-10;
          scroll[0].ratio=200;
          frame;
end
end

process proceso1()
begin
          ctype=c_scroll;
          cnumber=0;
          scroll[0].camera=id;
           graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
          x=320;y=240;
```

```
/*Estas dos líneas muestran que las coordenadas X e Y del proceso son relativas a la
esquina superior izquierda del gráfico del primer plano del scroll (y no de la pantalla
como hasta ahora) porque este proceso pertenece al scroll gracias a la línea
ctype=c_scroll. Si se observan valores negativos, es porque se ha salido fuera de los
límites del gráfico a causa de la repetición de éste en mosaico.*/
        write_var(0,20,10,4,x);
        write_var(0,20,20,4,y);
          loop
                    if(key(_up)) y=y-5;end
                    if(key(_down))y=y+5;end
                    if(key(_left))x=x-5;end
                    if(key(_right))x=x+5;end
                    frame;
          end
end

process proceso2()
begin
          ctype=c_scroll;
          cnumber=0;
          graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
          x=220;y=200;
        write_var(0,20,30,4,x);
        write_var(0,20,40,4,y);
          loop
                    if(key(_a))angle=angle+5000;end
                    if(key(_d))angle=angle-5000;end
                    if(key(_s))advance(5);end
                    frame;
          end
end
```

A partir del código anterior, podemos irlo ampliando y modificarlo según nuestros intereses. Por ejemplo, haz un ejercicio: intenta hacer que cuando "proceso1()" colisione con "proceso2()", éste último cambie su posición a otras coordenadas (x,y) aleatorias. Es muy sencillo.

Ahora vamos a ver un par más de ejemplos de diferentes scrolls, para tener más seguridad en este tema:

El siguiente ejemplo muestra un scroll con sólo un gráfico (el de primer plano), el cual se irá moviendo autónomamente e irá cambiando su dirección de movimiento automáticamente cada cinco segundos de tal manera: dirección noroeste,norte,noreste,este,sureste,sur,suroeste,oeste. Además, si se apreta la tecla Enter, el scroll se parará durante tres segundos, para reanudarse automáticamente en acabar éstos.

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_timers";

process main()
private
//Cantidad horizontal y vertical que se moverá el scroll en cada frame, respectivamente
  int ixS,iyS;
/*Según el valor que tenga, el valor de ixS e iyS será diferente, y por tanto el
movimiento del scroll será en una dirección u otra*/
  int DirS=1;
  int fpg;
end
begin
  set_mode(640,480);
  fpg=load_fpg("scroll.fpg");
  write(0,160,100,4,"ENTER = Detiene Scroll durante 3 segundos:");
  start_scroll(0,fpg,1,0,0,3); //Comienza el programa con el scroll iniciado
  repeat
 /*Si pulso ENTER y el scroll está funcionando, lo paro, y pongo  en marcha el
temporizador que a los tres segundos lo reemprenderá*/
```

```
    if(key(_enter))
      stop_scroll(0);
      timer[0]=0;
/*Durante tres segundos justo después de haber parado el scroll, no pasa nada, con lo que
el scroll continúa parado. Pasado este tiempo, lo volvemos a iniciar.*/
      while(timer[0]<=300) frame; end
      start_scroll(fpg,0,1,0,0,3);
   end

 /*Sumo 1 al valor de esta variable, hasta un máximo de 8, que son las combinaciones que
se han definido en este ejemplo para mostrar el  movimiento del scroll en las 8
direcciones principales de las dos dimensiones*/
    DirS=(DirS+1)%8;
    switch(DirS)
      case 1: ixS=3;  iyS=0;  end
      case 2: ixS=3;  iyS=3;  end
      case 3: ixS=0;  iyS=3;  end
      case 4: ixS=-3; iyS=3;  end
      case 5: ixS=-3; iyS=0;  end
      case 6: ixS=-3; iyS=-3; end
      case 7: ixS=0;  iyS=-3; end
      case 8: ixS=3;  iyS=-3; end
    end

 /*Ponemos en marcha otro temporizador que hará moverse en una determinada dirección,
determinada por ixS y iyS (determinados a su vez por DirS), al scroll durante 5 segundos.
Al cabo de los cuales, durante otros 5 segundos se moverá en otra dirección, y así*/
    timer[1]=0;
    while(timer[1]<=500)
      scroll.x0=scroll.x0+ixS; scroll.y0=scroll.y0+iyS;
      if(key(_esc)) exit(); end
      if(key(_enter)) break; end
      frame;
    end
  until(key(_esc))
end
```

Otro ejemplo muy interesante es el que viene a continuación. En él se puede observar el uso de un scroll delimitado dentro de una región definida, de tamaño menor que la pantalla. Con este ejemplo se puede ver cómo se podría implementar un scroll sólo en una zona concreta de la pantalla en vez de en toda ella, algo muy útil para multitud de juegos (carreras, mapas en miniatura,etc). Para poderlo ejecutar, necesitarás crear un gráfico mayor que el tamaño de la pantalla y tendrás que guardarlo dentro del FPG que estamos usando en estos ejemplos, “scroll.fpg” con el código 003. La idea es que esta nueva imagen sea el fondo fijo puesto con **put_screen()** visible en toda la pantalla excepto en una zona de ésta, que corresponderá a una región definida con **define_region()** -con el número 1- dentro de la cual se habrá implementado un scroll con movimiento automático vertical del gráfico de primer plano (código 001) y con el gráfico de segundo plano quieto (código 002).

```
import "mod_key";
import "mod_scroll";
import "mod_screen";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_timers";

const
  ANCHOPANT = 640;
  ALTOPANT = 480;
  ANCHOSCROLL = 300;
  ALTOSCROLL = 300;
end

global
  int fpgId;
end
```

```
process main()
BEGIN
fpgId = load_fpg("scroll.fpg");
set_mode(ANCHOPANT,ALTOPANT);
put_screen(fpgId,3);
define_region(1,(ANCHOPANT-ANCHOSCROLL)/2,(ALTOPANT-ALTOSCROLL)/2,
ANCHOSCROLL,ALTOSCROLL);
scroller();
loop
        if(key(_esc)) exit(); end
          frame;
end
END

PROCESS scroller()
BEGIN
x =ANCHOSCROLL/2;
y =ALTOSCROLL/2;
start_scroll(0, fpgId, 1, 2, 1, 3);
scroll[0].camera = ID;
loop
          y = y + 5;
        frame;
end
END
```

Este código tiene varias cosas interesantes. Lo primero es que definimos una región con el código 1, cuya esquina superior izquierda es un punto tal que el centro de la región coincide con el centro de la pantalla (es fácil verlo si haces un pequeño diagrama). Seguidamente, después de definir la región, llamamos a un proceso "scroller()", que se encargará de poner en marcha el scroll. Y esto lo hace de una manera peculiar: fijarse primero que el quinto parámetro de **start_scroll()** es "1", lo cual indica que el scroll sólo será visible en esa región, pero lo más interesante está en la línea siguiente: *scroll[0].camera=ID;* . Lo que hacemos aquí, ya lo debes saber, es hacer que el scroll se mueva siempre que se mueva el proceso actual: si éste no se mueve, el scroll no se moverá (es lo que llamábamos un scroll "perseguidor"). Pues aprovechamos precisamente este hecho para escribir dentro del LOOP/END un incremento perpetuo en la posición vertical del proceso actual. Como dicho proceso no parará de moverse de forma indefinida, el scroll lo seguirá de forma indefinida, con lo que observaremos el efecto del movimiento contínuo del scroll. Fíjate que no hemos definido ningún graph para este proceso, por lo que el proceso es invisible, pero a pesar de eso, su posición vertical varía de igual manera, que es lo que nos interesa para lo que pretendemos.

Otro ejemplo de scrolls donde se implementan regiones es el siguiente. En él se divide la pantalla en dos partes, cada una de las cuales es un scroll diferente. Habrá dos procesos: el primero será la cámara del primer scroll y el segundo será la cámara del segundo, con la particularidad de que este segundo proceso estará permanentemente persiguiendo y rodeando al primer proceso, creando así un curioso efecto.

```
import "mod_text";
import "mod_grproc";
import "mod_video";
import "mod_map";
import "mod_screen";
import "mod_mouse";
import "mod_scroll";
import "mod_proc";
import "mod_key";
import "mod_draw";
import "mod_math";

process Main()
private
    int idscroll,idscroll2;
end
begin
    set_mode(640,480,32);

    //Genero la imagen de scroll
    idscroll=new_map(100,100,32);map_clear(0,idscroll,rgb(100,100,100));
    idscroll2=new_map(50,50,32);map_clear(0,idscroll2,rgb(200,200,200));
```

```
    map_put(0,idscroll,idscroll2,0,0);

    //Un scroll se verá en la mitad izquierda de la pantalla
    define_region (1,0,0,320,480) ;
    start_scroll(0,0,idscroll,0,1,15);
    //El otro en la mitad derecha
    define_region (2,320,0,640,480) ;
    start_scroll(1,0,idscroll,0,2,15);

    /*Al crear proceso1, hago que sea la cámara del scroll de la izquierda, y
al crear proceso2 hago que sea la cámara del scroll de la derecha. Además, hago
que el identificador de proceso1 (que de hecho está guardado en scroll[0].camera,
sea utilizado como parámetro de proceso2 para que éste pueda perseguirlo*/
    scroll[0].camera = proceso1();
    scroll[1].camera = proceso2(scroll[0].camera) ;

    repeat
        frame;
    until (key(_esc))

    stop_scroll(0) ;
    stop_scroll(1) ;
    exit();
end

process proceso1()
begin
    graph = new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
    ctype = c_scroll;
    cnumber=0; //Se verá en todos los scrolls existentes
    loop
        if (key(_right)) x = x+4 ; end
        if (key(_left))  x = x-4 ; end
        if (key(_down))  y = y+4 ; end
        if (key(_up))    y = y-4 ; end
        frame;
    end
end

process proceso2(int procaperseguir)
begin
    graph = new_map(30,30,32);map_clear(0,graph,rgb(255,0,255));
    ctype = c_scroll ;
    cnumber=0;
    loop
//Trayectoria circular alrededor del procaseguir (proceso1)
        angle=(angle+5000)%360000;
        x = procaperseguir.x + get_distx (angle, 80) ;
        y = procaperseguir.y + get_disty (angle, 80) ;
        frame;
    end
end
```

A parte de los campos que ya hemos visto de la estructura ***SCROLL*** (x0,y1,ratio,camera...), existen unos cuantos campos más que merece la pena conocer porque son muy interesantes y nos pueden ayudar a conseguir efectos realmente logrados en nuestros juegos. Uno de estos campos es el campo **"z"**, que como puedes intuir, indica la profundidad del scroll. Por defecto su valor es 512. Piensa que aunque los scrolls tengan gráfico de primer plano y de segundo, su Z es única, por lo que el nombre de "primer plano" y "segundo plano" no corresponde a la realidad estricta: ambos gráficos poseen a efectos prácticos la misma profundidad. Si hacemos por ejemplo que la Z de un proceso sea mayor que la del scroll, éste tapará al gráfico del proceso, ya que estaremos dibujando el fondo movible encima. Una aplicación práctica: si los gráficos del scroll poseen amplias áreas transparentes (por ejemplo, representan el cielo salpicado de nubes), con una Z de scroll pequeña se logrará que éste pase por encima de los gráficos de los procesos con Z mayor. Además, aparte de poder jugar entre las profundidades del scroll y la de los procesos también se puede jugar con las profundidades de diferentes scrolls (*SCROLL[0], SCROLL[1]*...) entre ellos, logrando efectos realmente muy conseguidos.

Un ejemplo muy sencillo:

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";

process main()
private
  int idfpg;
end
begin
  set_mode(640,480);
  idfpg=load_fpg("scroll.fpg");
  start_scroll(0,idfpg,1,0,0,2);
  scroll[0].z=-1;
  proceso1();
  loop
            if(key(_esc)) exit(); end
            scroll[0].y0=scroll[0].y0 + 1;
            frame;
  end
end

process proceso1()
begin
          graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
          x=320;y=240;
          loop
                    frame;
          end
end
```

Ya que la Z de los procesos por defecto es 0, sólo hemos tenido que modificar la Z del scroll poniéndola a -1 para que el scroll se pinte por encima del gráfico del proceso. Si el gráfico del scroll tuviera muchas zonas transparentes, se podría comprobar el efecto comentado de las "nubes del cielo": si no, el proceso no se verá porque quedará oculto bajo el scroll.

Jugando con distintas profundidades para múltiples scrolls y/o personajes, se pueden obtener efectos curiosos. Por ejemplo, en el siguiente código tenemos dos scrolls automáticos independientes, de un sólo gráfico cada uno (o sea, no tienen gráfico de segundo plano), los cuales están situados a distintas profundidades y van a velocidades diferentes. Al usar dos scrolls pero sólo de un plano, el rendimiento es más o menos el mismo que si fuese uno solo de dos planos. Además, tenemos tres procesos cuyas profundidades se pueden observar que son: por encima de ambos scrolls, en medio de ellos y por debajo de ambos scrolls.Dependiendo de las zonas transparentes que tengas en los gráficos de los dos scrolls, verás o no según qué procesos en cada momento.

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";

process main()
private
  int idfpg;
end
begin
set_mode(800,600);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 1);
scroll[0].z= 10;
start_scroll(1, idfpg, 2, 0, 0, 1);
scroll[1].z= 20;

proceso(120,140,1);
```

```
proceso(360,240,15);
proceso(620,340,25);
loop
    if(key(_esc)) exit();end
    scroll[0].x0=scroll[0].x0 +5;
    scroll[1].x0=scroll[0].x0 * 2; //El scroll 1 va el doble de rápido que scroll 0
    frame;
end
end

process proceso(x,y,z)
begin
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,0));
  loop
       frame;
  end
end
```

Podemos hacer el mismo ejemplo pero haciendo que un scroll sea "perseguidor" de un proceso y el otro de otro proceso, por ejemplo. En el código siguiente, al igual que el anterior, hay dos scrolls de un sólo fondo, a diferentes z, y tres procesos cuya z es: encima de los dos scrolls, entre uno y otro y debajo de los dos. La novedad está en que uno de esos procesos es la cámara de uno de los scrolls, y otro proceso es la cámara del otro. Pruébalo.

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_grproc";

process main()
private
   int idfpg;
end
Begin
set_mode (800,600);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 1);
scroll[0].z= 10;
start_scroll(1, idfpg, 2, 0, 0, 1);
scroll[1].z= 20;

scroll[0].camera=proceso1(120,140,1);
scroll[1].camera=proceso2(360,240,15);
proceso3(620,340,25);
loop
        if(key(_esc)) exit();end
        frame;
end
end

process proceso1(x,y,z)
begin
  ctype=c_scroll;
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
  loop
         if(key(_up))advance(5);end
         if(key(_left)) angle=angle+5000;end
         if(key(_right)) angle=angle-5000;end
         frame;
   end
end

process proceso2(x,y,z)
begin
  ctype=c_scroll;
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
```

```
  loop
        if(key(_s))advance(5);end
        if(key(_a)) angle=angle+5000;end
        if(key(_d)) angle=angle-5000;end
        frame;
  end
end

process proceso3(x,y,z)
begin
  ctype=c_scroll;
  graph=new_map(30,30,32);map_clear(0,graph,rgb(255,255,255));
  loop
        frame;
  end
end
```

Pero,¡ojo!, si pruebas el código anterior, verás que ocurre una cosa muy rara: ¡los procesos se ven duplicados! Parece muy extraño pero es lógico: si te fijas, no le hemos dicho a ningún proceso en qué scroll queremos que sea visible (con ***CNUMBER***), con lo que por defecto los procesos serán visibles en todos los scrolls que haya. Así que, lo que tienes que hacer es lo siguiente: en "proceso1()" tienes que añadir al principio la línea *cnumber=c_0;* en "proceso2()" tienes que añadir la línea *cnumber=c_1;* y en "proceso3()" *cnumber=c_0;* ó *cnumber=c_1;* tanto da. Lo que hacemos con esto es decir que "proceso1()" sólo se verá en el scroll 0, del cual es cámara -y como tiene una Z menor, estará por encima y se podrá ver-, que "proceso2()" sólo se verá en el scroll 1 de la misma manera, y que "proceso3()" se ha de ver o en el scroll 0 o en el 1 pero que esto es indiferente porque como su Z es mayor, no se va a ver en ninguno. Pruébalo ahora y verás que obtienes el efecto deseado.

Existen más campos interesantes de la estructura ***SCROLL***. Uno de ellos es **"follow".** Este campo es recomendable usarlo con scrolls que sólo tengan el gráfico de primer plano, (ninguno de segundo plano, para evitar posibles inconsistencias). Para explicar su significado, partamos de este código, ya visto antes:

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";

process main()
private
          int idfpg;
end
Begin
set_mode (800,600);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 1);
scroll[0].z= 10;
start_scroll(1, idfpg, 2, 0, 0, 1);
scroll[1].z= 20;
loop
    if(key(_esc)) exit();end
    scroll[0].x0=scroll[0].x0 +5;
    scroll[1].x0=scroll[0].x0 * 2; //El scroll 1 va el doble de rápido que scroll 0
    frame;
end
end
```

Aquí tenemos dos scrolls a distintas profundidades, que se mueven de manera que uno está siguiendo al otro a distinta velocidad. Pues bien, el campo **"follow"** sirve precisamente para esto de forma no "tan manual": es decir,sirve para hacer que un scroll determinado siga automáticamente -"follow" quiere decir "seguir" en inglés- a otro scroll "maestro", cuyo número será el valor que asignaremos a este campo. Es decir, que si escribimos *scroll[1].follow=0;* , lo que estaremos diciendo es que el scroll nº 1 seguirá el movimiento que haga el scroll 0, sea en la dirección que sea. Así pues, podríamos sustituir la línea del código anterior *scroll[1].x0=scroll[0].x0 * 2;* por esta nueva que acabamos de descubrir: *scroll[1].follow=0;* Pero, ¿y la velocidad relativa?En la línea antigua decíamos que el scroll nº1 fuera el doble de rápido que el scroll nº 0. ¿Cómo se especifica ahora eso? Pues con el mismo campo **"ratio"** que ya conocemos. Si te acuerdas cuando explicamos este campo, dijimos que lo que significaba era el tanto

por ciento de velocidad relativa que el segundo plano de un scroll tenía respecto el primer plano. Pues bien, en el momento que un scroll persiga a otro, su campo "ratio" deja de significar eso para significar el tanto por ciento de velocidad relativa que tiene el scroll perseguidor del scroll perseguido.

Haz una prueba:añade antes de la orden frame; del ejemplo anterior esta línea: *scroll[1].ratio=200;* Verás que el scroll 1 va el doble de rápido que el 0. Si en cambio, escribiéras scroll[1].ratio=10, verías que va diez veces más lento.

Otro ejemplo ilustrativo donde se puede apreciar el uso de **"follow"** más el uso de regiones lo podemos encontrar en este código. En él tenemos definidas dos regiones (más la pantalla), y en cada una de ellas definimos un scroll diferente, que además se moverán a diferentes velocidades relativas. Este efecto es muy interesante si se quiere implementar por ejemplo un paisaje donde a primer plano (en la parte inferior de la pantalla) se puedan ver los elementos más cercanos moviéndose con rapidez, a medio plano (en la parte media de la pantalla, en otra región) se puedan ver los elementos un poco más alejados moviéndose más lentamente y en el plano de fondo (en una parte más superior de la pantalla, en otra región) se visualicen los elementos más alejados y más lentos. Como remate en este supuesto, se podría incluir en la parte más superior de la pantalla un fondo quieto con put_screen. El resultado es muy convincente.

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_screen";
import "mod_draw";

process main()
private
          int idfpg;
end
begin
set_mode(640,480);
idfpg=load_fpg("scroll.fpg");

//La región 1 estará en la parte superior de la pantalla, y la 2 en la inferior
define_region(1,0,0,640,224);
define_region(2,0,300,640,180);

//Dibujo dos rectángulos delimitando visualmente las dos regiones
draw_rect(0,300,640,480);
draw_rect(0,0,640,224);

//Prueba a cambiar los valores del último parámetro de start_scroll por 1, a ver qué pasa
start_scroll(1,idfpg,1,3,1,0);
/*El scroll 0 sólo se verá fuera de las dos regiones, en medio de la pantalla. Si se
quiere que se vea en las otras regiones, se tendrá que disminuir su Z (p.ej:
scroll[0].z=-1;)*/
start_scroll(0,idfpg,1,0,0,0);
start_scroll(2,idfpg,1,2,2,0);

//Sitúo en distintas coordenadas iniciales el gráfico de scroll
scroll[0].x0=250;
scroll[2].x0=420;
scroll[1].x0=550;

scroll[2].follow=0;
scroll[1].follow=0;
scroll[2].ratio=50;
scroll[1].ratio=250;

mariposo();

repeat
if(key(_left))scroll[0].x0=scroll[0].x0+4;end
if(key(_right))scroll[0].x0=scroll[0].x0-4;end
frame;
```

```
until(key(_esc))
exit();
end

process mariposo()
begin
x=320;y=250;
graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,255));
/*Con esta línea, las coordenadas X e Y se referirán al scroll y no a la pantalla. Se ha
de tener en cuenta que si el scroll también se mueve (ya que usamos la misma tecla para
el movimiento de las dos cosas), esto hará que el proceso vaya a una velocidad bastante
elevada, lógicamente, porque hay que sumar los 5 píxeles de rigor a la nueva posición del
scroll, la cual también está cambiando a cada momento*/
ctype=c_scroll;
//Hacemos que el proceso sólo sea visible en la franja del medio de la pantalla (el
scroll 0),
cnumber=c_0;
loop
if(key(_left)) x=x-5;end
if(key(_right)) x=x+5;end
if(key(_up)) y=y-5;end
if(key(_down)) y=y+5;end
frame;
end
end
```

Otros campos muy interesantes de la estructura ***SCROLL*** son **"flags1"** y **"flags2"**. Su significado y sus posibles valores (incluyendo las constantes B_TRANSLUCENT, B_HMIRROR, B_NOCOLORKEY, etc) son los mismos que la variable local predefinida ***FLAGS***, pero referidos al gráfico de primer plano y al de segundo plano de un scroll, respectivamente. Con estos campos puedes obtener efectos muy logrados de transparencias o efecto espejo, por ejemplo.

La estructura ***SCROLL*** también dispone de los campos **"region1"** y **"region2"**, que representan la llamada "región de bloqueo" (el primero para el gráfico de primer plano y el segundo para el gráfico de segundo plano del scroll). Es decir,que si hacemos por ejemplo *scroll[0].region1=2;*, lo que estaremos haciendo es asignar la región número 2 como región de bloqueo del gráfico de primer plano del scroll 0. Y una región de bloqueo es simplemente una zona donde el scroll no se moverá mientras el proceso cámara permanezca dentro de ella. Por defecto, el campo region1 vale -1, que quiere decir que no hay ninguna región de bloqueo.

También existe el campo **"speed"**, que sirve para marcar la velocidad máxima a la que el gráfico de primer plano del scroll podrá moverse. Su valor por defecto es 0, que significa que no tiene límite de velocidad.

En el uso de scrolls también disponemos de una función más (relativamente avanzada) que no hemos comentado hasta ahora:

| **MOVE_SCROLL(NUMERO)** |
|---|
| Esta función, definida en el módulo "mod_scroll", requiere como parámetro el número de scroll de 0 a 9 que se indicó cuando se inició el scroll mediante la función **start_scroll(),** como primer parámetro.<br><br>Esta función se utiliza cuando el scroll se controla automáticamente, por haber definido el campo **"camera"** de la estructura scroll correspondiente con el identificador de un proceso.<br><br>El propósito es forzar a que se actualicen los valores ("**x0", "y0", "x1"** e **"y1"**) de dicha estructura; si no se utiliza esta función estos valores no se actualizarán hasta la próxima imagen del juego. Es decir, cuando un proceso necesita conocer antes de la próxima imagen el valor de las coordenadas de un scroll (normalmente para colocarse él en una posición acorde al movimiento del fondo) se debe hacer esto:<br><br>1. Se inicia el scroll con **start_scroll()**<br>2. Se crea el proceso que se utilizará como cámara y se pone su código identificador en el campo camera de la estructura ***SCROLL***<br>3. A este proceso se le debe poner una prioridad muy alta, para que se ejecute antes que el resto de los procesos (poniendo en su variable local ***PRIORITY*** un valor entero positivo como, por ejemplo, 100).<br>4. Justo antes de la sentencia FRAME del bucle del proceso usado como cámara se llamará a la función |

***move_scroll()***

De esta forma se garantizará que este proceso se ejecute el primero y, justo al finalizar, actualice los valores (x0, y0, x1 y y1) de la estructura scroll, de forma que el resto de los procesos puedan utilizar estas variables ya actualizadas.

El uso más generalizado de esta función es cuando en una ventana de scroll se quieren tener más de dos planos de fondo y, para ello, se crean una serie de procesos que simulen un tercer o cuarto plano, situando sus coordenadas en función de la posición exacta del scroll en cada imagen.

PARAMETROS:
INT NUMERO : Número identificador de scroll (0 a 9)

Un ejemplo:

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_timers";

process main()
private
  int DirS;
  int fpg;
  int ixS,iyS;
end
begin
  set_mode(640,480);
  fpg=load_fpg("scroll.fpg");
  start_scroll(0,fpg,1,0,0,3);
  /*La cámara del scroll será el programa principal, el cual  variará su X e Y más
adelante y forzará así a mover el scroll  mediante move_scroll para seguirlo.*/
  scroll[0].camera=id;
  DirS=7;
  repeat
    switch(DirS)
      case 1: ixS=3;  iyS=0;  end
      case 2: ixS=3;  iyS=3;  end
      case 3: ixS=0;  iyS=3;  end
      case 4: ixS=-3; iyS=3;  end
      case 5: ixS=-3; iyS=0;  end
      case 6: ixS=-3; iyS=-3; end
      case 7: ixS=0;  iyS=-3; end
      case 8: ixS=3;  iyS=-3; end
    end
    timer[1]=0;
    while(timer[1]<=50)
      x=x+ixS; y=y+iyS;
      move_scroll(0);
      frame;
      if(key(_esc)) exit(); end
      if(key(_enter)) break; end
    end
    DirS=(DirS+1)%8;
  until(key(_esc))
end
```

Finalmente, no hemos hablado de la utilización del ratón con los scrolls. En concreto, en los ejemplos anteriores siempre hemos usado el teclado para mover nuestro personaje, pero, ¿qué pasa si a éste lo queremos dirigir mediante el ratón? Pues que tenemos que tener cuidado con las coordenadas. Para empezar, prueba este ejemplo.

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
```

```
import "mod_mouse";
import "mod_text";

process main()
private
          int idfpg;
end
Begin
set_mode (800,600);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 3);
scroll[0].camera=proceso1(160,300) ;
grafrat();
loop
    frame;
    if(key(_esc)) exit();end
end
end

process proceso1(x,y)
begin
  ctype=c_scroll;
  cnumber=c_0;
  graph=new_map(30,30,32);map_clear(0,graph, rgb(0,255,255));
  write_var(0,100,100,4,x);
  write_var(0,100,120,4,y);
  write_var(0,100,140,4,mouse.x);
  write_var(0,100,160,4,mouse.y);
  loop
     if(key(_left)) x=x-5; end
     if(key(_right)) x=x+5; end
     if(key(_up)) y=y-5; end
     if(key(_down)) y=y+5; end
     frame;
  end
end

process grafrat()
begin
  graph=new_map(10,10,32);map_clear(0,graph,rgb(255,255,255));
loop
   x=mouse.x;
   y=mouse.y;
   frame;
end
end
```

Es fácil ver en el código anterior que las coordenadas del proceso y las del ratón van completamente a su aire y no coinciden en nada. A estas alturas ya deberías saber por qué: porque el proceso que representa el cursor del ratón no pertenece a ningún scroll y por tanto sus coordenadas son relativas a la pantalla. Así que lo único que tenemos que hacer es añadir, antes del loop de "grafrat", las líneas *ctype=c_scroll;* y *cnumber=c_0;*. Si vuelves a probar el código, ahora verás que las coordenadas coinciden, por lo que parece que estamos cerca de la solución: ahora sólo tenemos que cambiar la manera en que se mueva "proceso1()" para que cada vez que se clique con el botón derecho, ese proceso se dirija al punto donde se ha hecho el click. Es decir, tener algo como esto:

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_mouse";
import "mod_text";

process main()
private
   int idfpg;
end
```

```
Begin
set_mode (800,600);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 3);
scroll[0].camera=proceso1(160,300) ;
grafrat();
loop
    frame;
    if(key(_esc)) exit();end
end
end

process proceso1(x,y)
begin
  ctype=c_scroll;
  cnumber=c_0;
  graph=new_map(30,30,32);map_clear(0,graph, rgb(0,255,255));
  write_var(0,100,100,4,x);
  write_var(0,100,120,4,y);
  write_var(0,100,140,4,mouse.x);
  write_var(0,100,160,4,mouse.y);
  loop
     if(mouse.left)
         x=mouse.x;
         y=mouse.y;
     end
     frame;
  end
end

process grafrat()
begin
  graph=new_map(10,10,32);map_clear(0,graph,rgb(255,255,255));
  ctype=c_scroll;
  cnumber=c_0;
loop
   x=mouse.x;
   y=mouse.y;
   frame;
end
end
```

Aparentemente el ejemplo va bien, pero ¡ojo!, si el cursor se mueve más allá de las coordenadas de la resolución de la pantalla (en este caso, 800x600), éste deja de responder: ¡no podemos mover nuestro personaje más allá de las dimensiones de la pantalla, aunque el gráfico de scroll sea más grande! ¿Cómo solucionamos esto? Así (las modificaciones están en negrita):

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_timers";
import "mod_mouse";
import "mod_text";

global
          int idgrafrat;
end

process main()
private
          int idfpg;
end
Begin
set_mode (800,600);
idfpg=load_fpg("scroll.fpg");
```

```
start_scroll(0, idfpg, 1, 0, 0, 3);
scroll[0].camera=proceso1(160,300) ;
idgrafrat=grafrat();
loop
    frame;
    if(key(_esc)) exit();end
end
end

process proceso1(x,y)
begin
  ctype=c_scroll;
  cnumber=c_0;
  graph=new_map(30,30,32);map_clear(0,graph, rgb(0,255,255));
  write_var(0,100,100,4,x);
  write_var(0,100,120,4,y);
  write_var(0,100,140,4,mouse.x);
  write_var(0,100,160,4,mouse.y);
  loop
     if(mouse.left)
         x=idgrafrat.x;
         y=idgrafrat.y;
     end
     frame;
  end
end

process grafrat()
begin
  graph=new_map(10,10,32);map_clear(0,graph,rgb(255,255,255));
  ctype=c_scroll;
  cnumber=c_0;
loop
   x=scroll[0].x1+mouse.x;
   y=scroll[0].y1+mouse.y;
   frame;
end
end
```

Hemos hecho que "proceso1()" siga no al ratón en sí sino al proceso que representa gráficamente su cursor, y dicho proceso tendrá unas coordenadas relativas al scroll dadas, en todo momento, por la suma de la posición del cursor del ratón respecto la pantalla más la posición en ese momento del scroll respecto la pantalla (es una simple traslación de coordenadas).

Otro ejemplo similar al anterior sería el que viene: aquí tenemos un proceso que está permanentemente orientado con su ***ANGLE*** mirando al cursor del ratón, y si apretamos las teclas de los cursores, si irá moviendo lentamente hacia la posición que ocupe en ese momento dicho cursor. No necesitas ningún gráfico externo.

```
import "mod_key";
import "mod_scroll";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_grproc";
import "mod_mouse";

CONST
                    screenx = 1024;
                    screeny = 768;
END
GLOBAL
                    int vizierid;
                    int mapscroll;
END

process main()
BEGIN
```

```
set_mode(screenx,screeny);
mapscroll=new_map(200,200,32);map_clear(0,mapscroll,rgb(155,30,234));
start_scroll(0,0,mapscroll,0,0,3);
character();
vizierid = vizier();
loop
                    if (key(_esc)) exit(); fade_off(); end
                    frame;
end
END


PROCESS vizier()
BEGIN
graph=new_map(20,20,32);map_clear(0,graph,rgb(55,55,34));
ctype=c_scroll;
LOOP
                    x=scroll[0].x1+mouse.x;
                    y=scroll[0].y1+mouse.y;
                    FRAME;
END
END

PROCESS character()
BEGIN
x=screenx/2;
y=screeny/2;
graph=new_map(50,50,32);map_clear(0,graph,rgb(155,155,134));
ctype=c_scroll;
scroll.camera=id;
LOOP
                    angle = get_angle(vizierid);
                    IF (key(_up)) xadvance(angle,4); END
                    IF (key(_down)) xadvance(angle,-4);END
                    IF (key(_left)) xadvance(angle+90000,4); END
                    IF (key(_right))xadvance(angle-90000,4); END
                    FRAME;
END
END
```

Ya para acabar, podríamos haber modificado el código anterior para hacer que el proceso “characte()r” cambie su orientación siguiendo la posición del cursor del ratón de otra manera diferente (es cuestión de gustos). En vez de escribir la línea *angle = get_angle(vizierid);* podríamos haber puesto:

```
difmouse = mouse.x-bmousex;
angle=angle+difmouse*1000;
bmousex = mouse.x;
```

donde *difmouse* y *bmousex* son variables privadas del proceso “character()”. Pruébalo.

¿Y qué pasa si en cambio pones...?

```
difmouse=screenx/2-mouse.x;
angle=angle+difmouse*1000;
```

## *Trabajar con puntos de control

Los puntos de control son posiciones que puedes definir dentro de un gráfico (puedes definir hasta 999).

Uno de ellos es especial: el punto 0, que define el "centro virtual" de la imagen. Si no defines el punto 0, el centro virtual pasa a ser automáticamente el centro real de la imagen. En ocasiones viene bien redefinir ese punto 0 y ponerlo en otro sitio (por ejemplo entre los pies del personaje si haces un juego de plataformas) ya que la imagen se

dibujará poniendo ese punto 0 en la posición dada por las variables locales ***X*** e ***Y*** del proceso. Otra cosa a tener en cuenta es que se escala (***SIZE***) y se rota (***ANGLE***) usando de centro también ese punto 0.

La utilidad de los puntos de control (aparte del 0 y lo comentado) es mucha. Usando éstos puedes definir, por ejemplo, dónde quieres situar un arma en un personaje en todos sus movimientos. Luego sólo tienes que buscar el punto de control desde el código y situar el arma sin tener que meter tú las posiciones en el código a pelo.

Bennu dispone de una serie de funciones que nos facilitan por un lado la creación de puntos de control en los gráficos de nuestro juego, y por otro, nos permite la obtención de las coordenadas de un punto de control concreto de un gráfico en particular, para su posible uso en nuestro código. Estas funciones básicamente son cuatro: **set_center()**, **set_point(), get_point()** (definidas en el módulo "mod_map") y **get_real_point()** (definida en el módulo "mod_grproc").

**SET_CENTER ( LIBRERIA, GRAFICO, X, Y )**

Esta función permite cambiar el centro de un gráfico, especificando unas nuevas coordenadas para el mismo. Estas coordenadas se dan siempre en pixels dentro del gráfico, donde el pixel (0, 0) representa la esquina superior izquierda del gráfico y las coordenadas crecen abajo y a la derecha.

El centro es un punto dentro de los pixels del gráfico que se utiliza para:

*Especificar el significado de la coordenada (***X,Y***) cuando el gráfico se visualiza en pantalla. Es decir: si un gráfico se visualiza en la coordenada (10, 15) significa que se dibujará de tal manera que su centro ocupará dicho punto.

*Especificar el punto a partir de cual se rota el gráfico -por ejemplo, variando el valor de la variable ***ANGLE*** del posible proceso asociado a ese gráfico-.El centro no se desplaza, y el resto del gráfico gira a su alrededor.

*Especificar el punto a partir del cual se realizan las operaciones de espejo horizontal y vertical,por ej mediante variable ***FLAGS*** .

Cuando un gráfico se utiliza como puntero del ratón, usando la variable ***mouse.graph***, indica el punto "caliente" del mismo, que se usa para comprobar dónde está marcando el ratón. En el ejemplo de Drag&Drop mostrado en el apartado sobre el ratón, se utilizó este hecho.

Cuando un gráfico no tiene centro asignado (por ejemplo, porque se ha recuperado a partir de un fichero estándar como puede ser un PNG), se utiliza como tal su centro geométrico.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico dentro de la librería
INT X : Nuevo valor para la coordenada horizontal del centro
INT Y : Nuevo valor para la coordenada vertical del centro

En el siguiente ejemplo se muestra un posible uso de establecer el punto central en otro lugar diferente del centro geométrico de la figura. En este caso, pintamos un cuadrado azul y situamos su centro cerca de uno de sus vértices. El programa consiste en irlo rotando ininterrumpidamente. Se podrá apreciar entonces que el eje de giro no es el centro geométrico sino el punto marcado en blanco, que representa precisamente el punto central (punto de control 0). De similar manera podríamos haber demostrado que ese punto es el eje de espejado cuando se cambia el valor de la variable ***FLAGS*** a 1 ó 2.

```
import "mod_map";
import "mod_key";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_draw";

process main()
private
//Coords del punto central (punto 0).Se pueden cambiar sus valores para ver el efecto
  int pmx=10,pmy=10;
end
begin
```

```
  x=160;
  y=100;
  graph=new_map(50,50,32);
  map_clear(0,graph,rgb(0,0,255));
 map_put_pixel(0,graph,pmx,pmy,rgb(255,255,255));//Pinto el punto central para que se vea
  //Si queremos que el punto central se vea un poco más grande
  //drawing_map(0,graph);draw_box(pmx-2,pmy-2,pmx+2,pmy+2);
  set_center(0,graph,pmx,pmy);
  repeat
    angle=(angle+7000)%360000;
    frame;
  until(key(_esc))
end
```

**SET_POINT ( LIBRERIA, GRAFICO, NÚMERO, X, Y )**

Un gráfico cualquiera puede contener hasta 999 puntos de control de los cuales el punto de control 0 representa el centro del gráfico, y el resto están disponibles para el usuario.

Mediante esta función puede asignarse un punto de control cualquiera del gráfico. Las coordenadas de este punto se dan siempre en pixels dentro del gráfico, donde el pixel (0, 0) representa la esquina superior izquierda del gráfico y las coordenadas crecen abajo y a la derecha.

Si el gráfico no contenía puntos de control, o no contenía puntos de control suficientes, se añadirán los necesarios. Por ejemplo, si un gráfico no dispone de puntos de control y mediante esta función se añade el punto de control 20, los puntos de control 0 a 19 serán rellenados con el valor indefinido (-1, -1).

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG
INT GRÁFICO : Número de gráfico dentro de la librería
INT NUMERO : Número de punto de control (0 a 999)
INT X : Nuevo valor para la coordenada horizontal del punto
INT Y : Nuevo valor para la coordenada vertical del punto

A continuación se presenta un ejemplo en donde aparece visualizado en pantalla un cuadrado azul, en el cual podremos establecer los puntos de control que deseemos, donde deseemos. Con los cursores podemos cambiar el número de punto de control que queramos introducir (teclas “1” y “2”), y sus coordenadas dentro del cuadrado (cursores del teclado). Aparecerá un punto blanco indicando en todo momento la posición que tendría en cada instante el punto de control. Y éste se crea de forma efectiva pulsando ENTER.

```
import "mod_map";
import "mod_key";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_draw";
import "mod_text";

process main()
private
  int pmx,pmy; //Coordenadas de un punto de control
  int pdc; //Número de punto de control
end
begin
  graph=new_map(100,100,32);
  x=220;
  y=100;
  map_clear(0,graph,rgb(0,0,255));
  write(0,10,30,3,"Punto Nº:");
  write_var(0,85,30,3,pdc);
  write(0,10,40,3,"Coord. X:");
  write_var(0,85,40,3,pmx);
  write(0,10,50,3,"Coord. Y:");
```

```
  write_var(0,85,50,3,pmy);
  write(0,10,70,3,"1 > Nº Punto");
  write(0,10,80,3,"2 < Nº Punto");
  write(0,10,90,3,"UP > Coord. Y");
  write(0,10,100,3,"DOWN < Coord. Y");
  write(0,10,110,3,"LEFT > Coord. X");
  write(0,10,120,3,"RIGHT < Coord. X");
  write(0,10,170,3,"ENTER = Set Point");
  repeat
    if(key(_1) && pdc<999) pdc++; end
    if(key(_2) && pdc>0)   pdc--; end
    if(key(_up)   && pmy>0)   pmy--; end
    if(key(_down) && pmy<99)  pmy++; end
    if(key(_left)  && pmx>0)  pmx--; end
    if(key(_right) && pmx<99) pmx++; end
    if(key(_enter)) set_point(0,graph,pdc,pmx,pmy); end

/*Para que no salga pintada una línea marcando el recorrido del cursor, y solamente
aparezca el punto blanco marcando la posición actual*/
    map_clear(0,graph,rgb(0,0,255));
    map_put_pixel(0,graph,pmx,pmy,rgb(255,255,255));
    frame;
  until(key(_esc))
end
```

¿Qué pasa si pulsamos ENTER cuando el número del punto de control está establecido en 0?¿Tienes una explicación?

**GET_POINT ( LIBRERÍA, GRÁFICO, NÚMERO, &X, &Y )**

Un gráfico cualquiera puede contener un número indeterminado de puntos de control de los cuales el punto de control 0 representa el centro del gráfico, y el resto están disponibles para el usuario.

Mediante esta función puede consultarse un punto de control del gráfico. Las coordenadas obtenidas se dan siempre en pixels dentro del gráfico, donde el pixel (0, 0) representa la esquina superior izquierda del gráfico y las coordenadas crecen abajo y a la derecha.

Si el número de punto de control especificado queda fuera de rango o no está definido, la función devuelve 0 y las variables x e y no serán modificadas. En el resto de casos, x e y pasarán a contener las coordenadas del punto de control.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

INT NUMERO : Número de punto de control (0 es el centro)

POINTER INT X : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada horizontal del punto respecto la esquina superior izquierda del propio gráfico.

POINTER INT Y : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada vertical del punto respecto la esquina superior izquierda del propio gráfico.

VALOR RETORNADO: INT : 1 si el punto está definido, 0 en caso contrario

Un ejemplo bastante sencillo de entender de la función anterior es el siguiente (en donde además, se puede ver cómo se puede dibujar por pantalla un punto, cambiando su posición de forma aleatoria cada x tiempo, utilizando temporizadores):

```
import "mod_map";
import "mod_key";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_draw";
import "mod_text";
import "mod_timers";
import "mod_rand";
```

```
process main()
private
  int gpx,gpy; //Coordenadas de un punto de control
  int pdc; //Número de un punto de control
  int p;
end
begin
  graph=new_map(100,100,32);
  map_clear(0,graph,rgb(0,0,255));
  x=160;
  y=120;
  write(0,10,30,3,"Get Point:");
  write_var(0,85,30,3,pdc);
  write(0,10,40,3,"Coord. X:");
  write_var(0,85,40,3,gpx);
  write(0,10,50,3,"Coord. Y:");
  write_var(0,85,50,3,gpy);
//Establezco las coordenadas (0,0) del gráfico como las del punto de control 0
  set_point(0,graph,0,0,0);
/*Establezco de forma aleatoria  (pero dentro de las dimensiones del gráfico mostrado)
las coordenadas del resto de 999 puntos de control posibles. Estas coordenadas tienen su
origen en la esquina superior izquierda del propio gráfico. */
  from p=1 to 999;
    set_point(0,graph,p,rand(0,99),rand(0,99));
  end
/*Cada décima de segundo obtendré las coordenadas de un punto de control seleccionado al
azar. Además, visualizaré mediante un punto blanco su situación dentro del gráfico*/
  timer[0]=0;
  repeat
    if(timer[0]>100)
      pdc=rand(0,999);
      get_point(0,graph,pdc,&gpx,&gpy);
      map_clear(0,graph,rgb(0,0,255));
      map_put_pixel(0,graph,gpx,gpy,rgb(255,255,255));
      timer[0]=0;
    end
    frame;
  until(key(_esc))
end
```

**GET_REAL_POINT ( NUMERO, &X, &Y )**

Esta función permite obtener la posición *respecto la pantalla* de un punto de control del gráfico asociado al *proceso actual.* .

El segundo y tercer parámetro de esta función pasarán a contener las coordenadas del punto de control del proceso actual, respecto la pantalla.

El valor de las variables locales ***X*** e ***Y*** contienen las coordenadas horizontal y vertical respectivamente del punto de control 0 del gráfico del proceso actual. Por tanto, la orden **get_real_point(0,&x,&y);** sería equivalente a consultar directamente el valor de dichas variables.

PARÁMETROS:

INT NUMERO : Número de punto de control (0 es el centro)

POINTER INT X : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada horizontal del punto respecto la esquina superior izquierda de la pantalla.

POINTER INT Y : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada vertical del punto respecto la esquina superior izquierda de la pantalla.

VALOR RETORNADO: INT : 1 si el punto está definido, 0 en caso contrario

El ejemplo siguiente es el mismo que utilizamos para mostrar el uso de **get_point()**, pero añadiéndole (y por tanto, pudiendo comparar los datos devueltos) una línea que ejecuta **get_real_point()** -y otra mostrando por pantalla

lo que esta función devuelve-, pudiendo observar así la diferencia de resultado entre ambas funciones. Básicamente, la diferencia está en que a pesar de devolver datos sobre el mismo punto la primera toma el origen de coordenadas del gráfico que se indica, y la segunda toma el origen de coordenadas de la pantalla, haciendo referencia al gráfico que esté asociado al proceso actual

```
import "mod_map";
import "mod_key";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_draw";
import "mod_text";
import "mod_timers";
import "mod_rand";
import "mod_grproc";

process main()
private
  int rgpx,rgpy; //Coordenadas de un pto de control respecto la pantalla (get_real_point)
  int gpx,gpy; //Coordenadas de un punto de control
  int pdc; //Número de un punto de control
  int p;
end
begin
  graph=new_map(100,100,32);
  map_clear(0,graph,rgb(0,0,255));
  x=160;
  y=120;
  write(0,10,30,3,"Get Point:");
  write_var(0,100,30,3,pdc);
  write(0,10,40,3,"Coord. Real X:");
  write_var(0,100,40,3,rgpx);
  write(0,10,50,3,"Coord. Real Y:");
  write_var(0,100,50,3,rgpy);
  write(0,10,60,3,"Coord. X:");
  write_var(0,100,60,3,gpx);
  write(0,10,70,3,"Coord. Y:");
  write_var(0,100,70,3,gpy);
//Establezco las coordenadas (0,0) del gráfico como las del punto de control 0
  set_point(0,graph,0,0,0);
/*Establezco de forma aleatoria  (pero dentro de las dimensiones del gráfico mostrado)
las coordenadas del resto de 999 puntos de control posibles. Estas coordenadas tienen su
origen en la esquina superior izquierda del propio gráfico. */
  from p=1 to 999;
    set_point(0,graph,p,rand(0,99),rand(0,99));
  end
/*Cada décima de segundo obtendré las coordenadas de un punto de control seleccionado al
azar. Además, visualizaré mediante un punto blanco su situación dentro del gráfico */
  timer[0]=0;
  repeat
    if(timer[0]>100)
      pdc=rand(0,999);
      get_point(0,graph,pdc,&gpx,&gpy);
      get_real_point(pdc,&rgpx,&rgpy);
      map_clear(0,graph,rgb(0,0,255));
      map_put_pixel(0,graph,gpx,gpy,rgb(255,255,255));
      timer[0]=0;
    end
    frame;
  until(key(_esc))
end
```

Seguramente estarás deseando encontrar alguna aplicación gráfica, sencilla y cómoda que nos permita crear, manipular y consultar puntos de control de un gráfico sin tener que recurrir a programar "a pelo" en Bennu. Afortunadamente, la mayoría de editores FPG (el "FPGEdit", el editor de Prg, el "Smart FPG Editor") permiten todo esto y bastante más. Por ejemplo, si estás usando el "FPGEdit", abre un archivo FPG cualquiera y selecciona alguna de las imágenes que están contenidas en él. Pulsa en el botón "Edit" de la barra de herramientas de la zona inferior: verás

que aparece un cuadro ya conocido, donde podemos ver cierta información sobre la imagen (las dimensiones, el identificador del gráfico dentro del FPG,etc). Pero no sé si te habías fijado que también hay un botón en ese cuadro, arriba a la izquierda, que pone "Puntos de control". Si clicas allí verás como el contenido del cuadro cambia y aparecen un montón de opciones referentes a la manipulación de puntos de control. Lo más básico que podemos hacer allí es añadir las coordenadas dentro de la imagen del nuevo punto de control que queremos (con las cajas de texto correspondientes, o si quieres, con la ayuda del cuadradito formado por los botones radio) y crearlo pulsando sobre el botón con el icono de la flecha apuntando a la izquierda. Como consecuencia, ese punto aparecerá en la lista de la izquierda. Para eliminar un punto, simplemente hay que seleccionarlo de allí y pulsar sobre el botón con el icono de la flecha apuntando a la derecha. En principio el número del punto de control se asigna automáticamente de forma consecutiva dependiendo de los puntos que ya haya definidos. Puedes investigar por tu cuenta las otras opciones que aporta el programa: todas son interesantes, y sobre todo, bastante más rápidas que hacerlo uno mismo con código. Los demás editores FPG son similares en funcionalidad y comportamiento.

Como ejemplo práctico de utilización de los puntos de control, vamos a considerar el caso de una típica barra de energía. Imagínate que estamos programando un juego donde nuestro personaje se enfrenta a muchos enemigos que le atacan constantemente y hacen que su energía disminuya en cada envestida. Un caso similar ya nos lo encontramos en el capítulo-tutorial del juego del matamarcianos. En aquel momento se hizo uso de las regiones. Ahora vamos a utilizar otra técnica para obtener un resultado idéntico, con un ejemplo simple de gestión de puntos de control. Para ello, dibuja primero un gráfico rectangular de unos 100x30 píxeles que nos permita visualizar la energía que tiene en cada momento nuestro personaje. La idea es que a medida que la energía disminuya, esta barra se ha de hacer más y más corta. En el código que presento a continuación, se puede observar que tenemos un proceso "personaje()" (que resulta ser un cuadrado inmóvil) y un proceso "barra()" que es el responsible de mostrar la barra de energía de este personaje. En vez de complicar el ejemplo con procesos enemigos, se ha optado por hacer que nuestro personaje decremente su vida en una unidad cada vez que se pulse la tecla "e". Así pues, el enemigo de nuestro personaje será esta tecla. Aquí está el programa:

```
import "mod_map";
import "mod_key";
import "mod_proc";
import "mod_key";
import "mod_draw";

Global
   int vidapersonaje=100;
End

process main()
Begin
personaje();
barra();
Loop
    if(key(_esc)) exit();end
    Frame;
End
End

Process personaje()
Begin
graph=new_map(50,50,32);map_clear(0,graph,rgb(0,255,255));
x=160;y=120;
Loop
     If(key(_e)) vidapersonaje--; end
     Frame;
End
End

process barra()
Begin
graph=new_map(100,30,32);map_clear(0,graph,rgb(0,255,0));
x=160;y=50;
Loop
     if (vidapersonaje<size_x) size_x--; end
     Frame;
End
End
```

La línea clave está en *IF (vidapersonaje<size_x) size_x--; END.* Recuerda que la variable local predefinida ***SIZE_X*** establece el tanto por ciento del tamaño del gráfico en su coordenada horizontal, y por defecto vale 100. Bien. Para que esta línea sea útil hemos utilizado dos premisas: 1º) en cada ataque enemigo (cada pulsación de "e") la energía disminuye una unidad; 2º) la energía inicial es 100. La primera premisa es importante porque en la línea del if también disminuimos ***SIZE_X*** en una unidad: para que el efecto del decrecimiento de la barra esté acorde con la energía real que tiene nuestro personaje, ambas cosas (energía y barra) han de decrecer al mismo ritmo. Y esto tiene mucho que ver también con la segunda premisa: los valores iniciales de la energía y de ***SIZE_X*** son iguales, para hacer así que cuando se tenga la mitad de energía (50), la barra tenga la mitad del tamaño original (SIZE_X=50) y así. Es evidente ambas cosas que podrían tener valores iniciales diferentes (y decrementos diferentes también), pero entonces se complicaría la cosa: para empezar, la condición del if se tendría que cambiar, porque si te fijas, tal como está lo que hace es precisamente suponer que *vidapersonaje* y ***SIZE_X*** tienen valores que van a la par, y cuando *vidapersonaje* decrece una unidad y se queda por debajo de ***SIZE_X***, entonces esta variable es la que decrece una unidad inmediatamente después.

Bien, si has probado de ejecutar este código, verás que la barra decrece,sí, pero no como te gustaría seguramente. Decrece por los dos lados por igual (derecho e izquierdo), y lo que sería más razonable es que de un lado permaneciera fija y sólo decreciera del otro -normalmente el lado derecho-.¿Cómo solucionamos esto? Con los puntos de control. Más exactamente, lo que haremos será modificar la posición del punto de control 0 -es decir, el centro virtual- para que deje de estar en el centro real de la imagen y pase a estar en el extremo izquierdo de la barra (y centrado verticalmente, eso sí). Haciendo esto, conseguiremos que la barra sólo disminuya por el lado contrario, ya que la posición de su centro permanecerá inalterable. ¿Qué tenemos que hacer, pues? Escribe la siguiente línea dentro del proceso "barra()", justo después del LOOP/END:

```
set_center(0,graph,0,15);
```

Fíjate que hemos situado el punto de control 0 del gráfico dado por ***GRAPH*** en la coordenada horizontal 0 y la coordinada vertical 15 (30/2, donde 30 es el alto de la imagen). Si ahora pruebas el programa, verás que la barra ya disminuye su valor tal como queríamos (aunque verás que aparece un poco desplazada hacia la derecha, precisamente porque hemos alterado su centro virtual que ha dejado de ser su centro real)

No obstante, una de las aplicaciones más habituales de los puntos de control es situar un proceso en unas determinadas coordenadas relativas a la posición de otro proceso. Por ejemplo, situar una espada sobre la mano de nuestro protagonista: se podrían hacer complejos cálculos para ubicar la espada teniendo en cuenta la posición del personaje, su tamaño, etc...pero es mucho más fácil definir un punto de control allí donde se desee colocar la espada, y directamente colocarla allí, independientemente de las coordenadas concretas que tenga ese punto en cada momento.

En el siguiente ejemplo se muestra esta idea. El código busca la posición de la mano (punto control 1 ), la posición del centro virtual del personaje (punto control 0), y coloca la espada relativamente respecto a ambos puntos de control, teniendo en cuenta el flags, también. El código básicamente hace todo menos la rotación de angulos, para eso habría que ir comprobando el gráfico y probando el ángulo que quedaría mejor. También se puede jugar con Z usando "profundidad", para que la espada se vea por delante o detras del gráfico según la animación.

Para una buena sincronización es importante que el proceso "espada()"se ejecute después de "prota()" , modificando su valor de ***PRIORITY***.

```
Import "mod_map";
Import "mod_key";
Import "mod_proc";
Import "mod_map";
Import "mod_screen";

process main()
begin
prota();
repeat
        frame;
until(key(_esc))
exit();
end

process prota()
```

```
begin
graph=new_map(100,100,32);map_clear(0,graph,rgb(255,255,255));
x=160;y=120;
/*Establezco el punto (70,70) de dentro de la imagen de este proceso para ubicar el punto
de control 1, el cual puede representar la mano que empuña la espada*/
set_point(0,graph,1,70,70);
espada();
loop
          if(key(_left)) flags=2; x=x-10; end
          if(key(_right)) flags=0;x=x+10; end
          if(key(_up)) y=y-10; end
          if(key(_down)) y=y+10; end
          frame;
end
end

process espada()
private
          int punto_controlX,punto_controlY;
          int punto_origenX, punto_origenY;
          int profundidad=1;
end
begin
graph=new_map(20,20,32);map_clear(0,graph,rgb(255,255,0));
loop
          if(!exists(father)) return; end // por si acaso
/*Obtengo el punto de control 1 del proceso prota (puede ser por ejemplo, la mano que
empuña la espada)*/
          get_point(father.file,father.graph,1,&punto_controlX,&punto_controlY);
//Obtengo el punto de control 0 (el centro) del proceso prota
          get_point(father.file,father.graph,0,&punto_origenX,&punto_origenY);
/*Controlo la orientación izquierda-derecha del gráfico, y coloco horizontalmente la
espada en consecuencia*/
          flags=father.flags;
          if(flags==0)
                    x=(father.x-punto_origenX)+punto_controlX;
          else
                    x=(father.x+punto_origenX)-punto_controlX;
          end
//Coloco verticalmente la espada
          y=(father.y-punto_origenY)+punto_controlY;
//Para que la espada se vea por encima del protagonista
          z=father.z-profundidad;
          frame;
end
end
```

## *Trabajar con sonido

Antes de poder usar los maravillosos efectos especiales de sonido que hemos preparado (explosiones, disparos, choques,etc) tenemos que decirle al programa que los cargue en memoria. Y esto se hace con la instrucción **load_wav()**. Esta función se encarga, como su nombre indica, de cargar en memoria (que no de reproducir nada: sólo carga) un fichero de audio con formato wav.

En el Apéndice B de este texto comentamos que este formato contiene una grabación digital de audio sin comprimir, y que por tanto ocupa mucho espacio en disco –y en memoria, claro-.Por lo tanto, los archivos wav son recomendables sólo cuando se quieren utilizar sonidos realistas pero relativamente cortos de duración, como puede ser una explosión, un choque…pero no para poner una canción, por ejemplo.De hecho, para poner música en un juego –que no sonidos concretos- existe una manera distinta de trabajar, de la cual hablaremos dentro de nada. También hay otro uso de los ficheros wav, y es el de poner un sonido que se repita constantemente a lo largo de un período de tiempo, ya que la carga del wav en memoria sólo se realiza la primera vez, y a partir de ésta se puede ir repitiendo la emisión de ese sonido sin problemas. También lo veremos.

Una vez cargado el sonido, entonces se procederá a reproducirlo, con la función **play_wav()**. Podremos parar la reproducción con **stop_wav()**, pausarla con **pause_wav()** y reemprenderla con **resume_wav()** y varias cosas más como cambiar el volumen o el balance del sonido con las funciones que veremos a continuación.

Todas las funciones comentadas en este apartado, si no se indica explícitamente lo contrario, están definidas en el módulo "mod_sound".

**LOAD_WAV("FICHERO")**

Esta función carga un fichero en formato WAV de disco y lo almacena en memoria para reproducirlo en el futuro mediante la función **play_wav()** . El valor que devuelve la función es un número entero que representa el recurso en memoria, o -1 si no fue posible cargar el fichero o no estaba en un formato correcto.

Si no se desea utilizar por más tiempo, puede descargarse de memoria el efecto de sonido empleando la función **unload_wav()**

PARAMETROS: STRING FICHERO **:** Nombre del fichero Wav

VALOR RETORNADO: INT: Identificador del sonido

**PLAY_WAV(WAV, REPETICIONES)**

Reproduce un efecto de sonido recuperado previamente de disco mediante la función **load_wav()**

Es posible reproducir varias veces un mismo efecto de forma simultánea. Esta función devuelve un identificador del "canal de sonido" empleado para reproducir el efecto de sonido. Pueden existir hasta un máximo de 32 canales de sonido disponibles para efectos, y esta función puede elegir cualquiera de ellos que esté libre de forma dinámica. Es decir, el programador no necesita especificar qué canal se utilizará para reproducir un sonido: es el propio Bennu el que se encarga de escoger el primer canal disponible que encuentre.

Si no es posible reproducir un sonido (por ejemplo, porque haya una gran cantidad de efectos sonando y ocupen los todos los canales disponibles) esta función devolverá -1.

El identificador de canal devuelto por esta función puede usarse con **is_playing_wav(), set_wav_volume()** , y demás funciones de sonido.

El efecto de sonido puede reproducirse una sola vez, si se pasa como parámetro de repeticiones 0, el número concreto de repeticiones indicado, o bien indefinidamente si este parámetro es -1. Hay que tener en cuenta que si el sonido se reproduce indefinidamente, el canal quedará ocupado hasta que el sonido se detenga con **stop_wav()**

PARAMETROS:

INT WAV : Identificador del sonido devuelto por **load_wav()**
INT REPETICIONES : Número de repeticiones (0=1 vez; -1= infinitas veces)

VALOR RETORNADO: INT: Número de canal por donde sonará el efecto

**STOP_WAV(CANAL)**

Detiene la reproducción de cualquier sonido que esté activo a través de un canal determinado. Para saber qué canal corresponde a un efecto de sonido concreto, debes guardar el valor devuelto por **play_wav()**

PARAMETROS:

INT CANAL : Número de canal devuelto por **play_wav()**

**PAUSE_WAV(CANAL)**

Detiene temporalmente la reproducción de cualquier sonido que esté activo a través de un canal determinado, hasta que vuelva a reanudarse la reproducción empleando **resume_wav()** . Para saber qué canal corresponde a un efecto de sonido concreto, debes guardar el valor devuelto por **play_wav()**

PARAMETROS:

INT CANAL : Número de canal devuelto por **play_wav()**

**RESUME_WAV(CANAL)**

Reanuda la reproducción del sonido que estaba activo a través de un canal determinado en el momento que se detuvo la reproducción empleando **pause_wav()**. Para saber qué canal corresponde a un efecto de sonido concreto, debes guardar el valor devuelto por **play_wav()**

PARAMETROS:

INT CANAL : Número de canal devuelto por **play_wav()**

**IS_PLAYING_WAV(CANAL)**

Esta función comprueba si cualquier efecto de sonido reproducido anteriormente mediante **play_wav()** , todavía está sonando, devolviendo en este caso 1. En caso contrario esta función devolverá 0.

Hay que tener en cuenta que el canal podría ser reutilizado más tarde por otros efectos de sonido (si no está reservado y por tanto es asignado dinámicamente), por lo que es conveniente no seguir haciendo la comprobación una vez esta función ha devuelto 0.

PARAMETROS:

INT CANAL : Número de canal devuelto por **play_wav()**

**SET_CHANNEL_VOLUME(CANAL,VOLUMEN)**

Especifica el volumen general del canal de sonido, entre 0 (silencio) y 128 (volumen máximo).

Si el canal especificado es -1, afecta al volumen de todos los canales.

A parte de este nivel de volumen por cada canal, el volumen de cada efecto de sonido que se reproduzca podrá ser regulado independientemente por **set_wav_volume()**. De tal manera, el volumen final de un efecto de sonido vendrá dado por estas dos funciones: **set_channel_volume()** establece el volumen base del canal y **set_wav_volume()**, a partir de ese volumen base, lo modifica para un efecto de sonido en concreto.

PARÁMETROS :

INT CANAL : Número de canal devuelto por **play_wav()**
INT VOLUMEN : Volumen deseado (0-128)

**SET_WAV_VOLUME(WAV,VOLUMEN)**

Cambia el volumen de un efecto de sonido cargado anteriormente mediante **load_wav()** , entre 0 (silencio) y 128 (volumen original del efecto).

El volumen final de reproducción del efecto será este modificado por el valor de volumen de reproducción asignado al canal por el que esté con **set_channel_volume()**

PARÁMETROS :

INT WAV : Identificador del efecto devuelto por **load_wav()**
INT VOLUMEN : Volumen deseado (0-128)

**UNLOAD_WAV(WAV)**

Elimina un efecto de sonido de memoria. Es importante que el efecto no se esté reproduciendo en el momento de descargarlo de memoria, por lo que debes asegurarte de ello mediante la función **is_playing_wav()**

Normalmente sólo es preciso descargar de memoria efectos de sonido contínuos, que ocupan más espacio, como el

motor de un coche o una lluvia de fondo.

PARÁMETROS :

INT WAV : Identificador del efecto devuelto por **load_wav()**

Un ejemplo trivial del uso de estas funciones sería por ejemplo crear un mando con los cursores del teclado que controle en tiempo real el volumen de un sonido que se está reproduciendo en este momento:Necesitarás un archivo de sonido llamado “a.wav”:

```
import "mod_key";
import "mod_text";
import "mod_sound";

process main()
private
  int wav;
  int vol=64;
end
begin

  write(0,10,90,3,"-> = UP Volumen...");
  write(0,10,100,3,"<- = DOWN Volumen...");
  write(0,10,110,3,"Volumen Actual... ");
  write_var(0,124,110,3,vol);
  wav=load_wav("a.wav");
  play_wav(wav,-1);
  repeat
    if(key(_left)  && vol>0)   vol=vol-8; end
    if(key(_right) && vol<128) vol=vol+8; end
    set_wav_volume(wav,vol);
    frame;
  until(key(_esc))
end
```

Es posible que a veces nos interese que **play_wav()** no pueda utilizar todos los canales disponibles sin restricción, sino que tengamos una serie de canales reservados para un uso específico que no entren en la “elección dinámica” hecha por Bennu a la hora de buscar canales libres. La idea es utilizar una serie de canales reservados para fijar en ellos determinados efectos de sonido que sabemos que siempre sonarán por ese canal concreto, y no de forma aleatoria por el primer canal que se encuentre libre. El primer paso es, pues, definir cuántos canales, de los que hay disponibles, estarán reservados para uso personal, sin que Bennu pueda trabajar en ellos por su propia cuenta.

Pero para ello nos encontramos con un problema previo: ¿cuántos canales hay disponibles por defecto en Bennu? Hemos dicho que se pueden disponer hasta 32 como máximo, pero por defecto sólo son 8. Si deseamos aumentar (o disminuir) el número de canales a usar, deberemos utilizar la variable global predefinida ***SOUND_CHANNELS,*** asignándole el número de canales que se quieren utilizar (de 1 hasta 32).

Bien, una vez tengamos los canales que deseemos, de éstos podremos reservar unos cuantos para ser gestionados por nosotros de forma personalizada con la función **reserve_channels()**.

**RESERVE_CHANNELS(NCANALES)**

Esta función reserva el número de canales especificado como parámetro (comenzando por el primero, el número 0). Es decir, si se escribe *reserve_channels(6);*, se estará reservando del canal 0 al 5. Esta función devuelve la cantidad de canales efectivamente reservados, ó -1 si ha ocurrido algún error.

PARAMETROS:

INT NCANALES : Número de canales reservados para uso personalizado (no dinámico)

Una vez reservados el número de canales deseado, para utilizar en concreto uno de éstos , se utilizará la función **play_wav()** ya conocida, pero con la diferencia de tener añadido un tercer parámetro que es precisamente el

número de canal específico por el que queremos que suene nuestro archivo wav.

| **PLAY_WAV(WAV, REPETICIONES, CANAL)** |
|---|
| Funciona exactamente igual que la función **play_wav()** vista anteriormente, pero incluye un tercer parámetro extra que sirve para especificar el canal concreto (reservado previamente para nuestro uso con **reserve_channels()**) que se desea utilizar específicamente para reproducir el sonido. Por tanto, la gran diferencia entre las dos funciones **play_wav()**es que la primera utilizará un canal a priori desconocido (el que Bennu encuentre libre) para reproducir un sonido y la segunda utilizará el canal que nosotros hemos elegido explícitamente (siempre y cuando esté reservado).<br><br>Si como valor del tercer parámetro establecemos un -1, sería equivalente a la primera función **play_wav()**, es decir, no estamos especificando ningún canal concreto y delegamos su elección a la que haga Bennu por su cuenta.<br><br>El significado de los dos primeros parámetros es idéntico al de la función **play_wav()** anterior. Al igual que pasaba en ella, si se especifica un -1 como número de repeticiones, estaremos creando un loop infinito en la reproducción del efecto. Y al igual que pasaba en la función **play_wav()** anterior también, el valor devuelto será -1 si ha ocurrido algún error o bien el número de canal utilizado para la reproducción del sonido, que en este caso debería coincidir con el valor que hayamos puesto como tercer parámetro (a no ser que éste sea -1).<br><br>PARAMETROS:<br><br>INT WAV : Identificador del sonido devuelto por **load_wav()**<br>INT REPETICIONES : Número de repeticiones (0=1 vez; -1= infinitas veces)<br>INT CANAL: Número de canal (reservado previamente) por donde se desea que suene el efecto<br><br>VALOR RETORNADO: INT: Número de canal por donde sonará el efecto |

Un detalle MUY importante es tener en cuenta que tanto la variable ***SOUND_CHANNELS*** como la función **reserve_channels()** se han de utilizar SIEMPRE ANTES DE CARGAR CUALQUIER SONIDO. Es decir, siempre antes de utilizar ningún **load_wav()**. Si no se hace así, lo que hayamos establecido como número de canales y/o número de canales reservados no se tendrá en cuenta. Cuidado con esto.

Veamos un ejemplo (necesitarás un fichero de audio llamado "a.wav") donde se muestren las diferencias entre las dos funciones **play_wav():**

```
import "mod_key";
import "mod_text";
import "mod_sound";

process main()
Private
  int idwav;
  int idcanal;
end
begin
  sound_channels=32; //Se pueden usar 32 canales
  reserve_channels(21); //Se puede usar de forma no dinámica desde el canal 0 hasta el 20
  idwav=load_wav("a.wav");
  repeat
    if(key(_a))
            while(key(_a)) frame; end
            idcanal=play_wav(idwav,0);
            delete_text(0);
/*Debería salir cada vez un número de canal diferente (el primero que se encuentre libre)
pero siempre mayor de 20, que es el último canal reservado*/
            write(0,0,0,0,idcanal);
    end
    if(key(_s))
            while(key(_s)) frame; end
            idcanal=play_wav(idwav,0,4);
            delete_text(0);
//Debería salir 4 siempre. Si el canal no hubiera estado reservado antes,daría error (-1)
            write(0,0,0,0,idcanal);
```

```
    end
    frame;
  until(key(_esc))
  unload_wav(idwav);
end
```

Otras funciones muy interesantes son:

| **SET_DISTANCE(CANAL, DISTANCIA)** |
|---|
| Esta función permite alterar el sonido de un canal determinado simulando un alejamiento del mismo, a partir de una distancia al jugador. La distancia 0 deja el sonido como estaba al principio, mientras que una distancia de 255 hace el sonido inaudible.<br><br>Esta función filtra el sonido de un canal sin hacer uso de las características avanzadas de algunas tarjetas de audio. Si se desea una emulación algo más sofisticada, es mejor usar **set_position()**<br><br>PARÁMETROS :<br>INT CANAL : Número de canal devuelto por **play_wav()**<br>INT VOLUMEN : Distancia del jugador (0-255) |

| **SET_POSITION(CANAL,ANGULO,DISTANCIA)** |
|---|
| Esta función permite alterar el sonido de un canal determinado simulando un posicionamiento en 3D del mismo, a partir de un ángulo de posición y una distancia al jugador. Los párametros (0,0) dejan el sonido como estaba al principio, mientras una distancia de 255 hace el sonido inaudible.<br><br>Esta función filtra el sonido de un canal de una forma algo lenta, sin hacer uso de las características avanzadas de algunas tarjetas de audio. Si sólo se desea una emulación más simple, es más rápido llamar a **set_distance()**.<br><br>PARÁMETROS :<br>INT CANAL : Número de canal devuelto por **play_wav()**<br>INT ANGULO : Ángulo respecto al jugador, en grado (0-360)<br>INT DISTANCIA: Distancia del jugador (0-255) |

| **SET_PANNING(CANAL,IZQUIERDO,DERECHO)** |
|---|
| Esta función cambia el balance de un canal. Es decir, esta función permite cambiar el posicionamiento izquierdo/derecho en stereo de un sonido, simplemente ajustando el volumen de cada uno de los dos subcanales. Si se desea dejar el sonido tal cual estaba en un principio, basta con hacer un **set_panning(255,255)**.<br><br>PARÁMETROS :<br>INT CANAL : Número de canal devuelto por **play_wav()**<br>INT IZQUIERDO : Volumen izquierdo (0-255)<br>INT DERECHO: Volumen derecho (0-255) |

| **REVERSE_STEREO(WAV, FLIP)** |
|---|
| Esta función invierte el posicionamiento de balanceo de un sonido estéreo (izquierdo<->derecho y derecho<->izquierdo), siempre y cuando el segundo parámetro sea diferente de 0. Si éste es igual a 0, restaura el balance original.<br><br>PARÁMETROS :<br>INT WAV : Identificador del sonido devuelto por **load_wav()**<br>INT FLIP : Si es igual a 0 no se producirá la inversión. Si es diferente, sí se producirá. |

Las dos funciones anteriores necesitan utilizar el sonido en modo Estéreo para poder realizar su función correctamente. ¿Cómo se puede decidir desde el código Bennu si un sonido sonará en modo Mono o en modo Estéreo? Pues con la variable predefinida global ***SOUND_MODE***, la cual puede tener dos valores: MODE_MONO (equivalente

al valor numérico 0) ó MODE_STEREO (equivalente al valor numérico 1). Estableciendo el valor deseado (¡siempre antes de cualquier carga de cualquier sonido!) ya podremos usar mono ó estéreo según nuestra conveniencia. Por defecto, esta variable vale 1.

Ahora que hablamos de variables globales predefinidas de sonido, comentar que existe otra, llamada ***SOUND_FREQ***, la cual tiene el valor predefinido de 22050, pero que puede tener otros dos valores más: 11025 y 44100. Esta variable sirve para establecer la frecuencia de muestreo a la que sonará el audio: a mayor frecuencia mayor calidad.

Un ejemplo muy simple de las funciones de sonido recién comentadas lo podemos ver aquí. En este código se crea un proceso, el cual mientras esté vivo mostrará por pantalla un rectángulo de color amarillo. Y este proceso estará vivo el tiempo que dure en sonar un archivo wav (llamado "a.wav"). El usuario podrá pausar y reemprender la ejecución del sonido con la tecla "s" y "r" respectivamente.

Notar que todas las funciones utilizadas en el ejemplo excepto **play_wav(), set_wav_volume()** y **unload_wav()** tienen como parámetro el identificador de canal, y no el identificador del archivo wav.

```
import "mod_key";
import "mod_text";
import "mod_sound";
import "mod_map";

global
   int idwav;
end

process main()
begin
/*La carga de los sonidos se puede hacer al principio del programa, al igual que se hace
con los fpgs.*/
idwav=load_wav("a.wav");
miproceso();
while(!key(_esc))
          frame;
end
end

process miproceso()
private
  int idcanal;
end
begin
graph=new_map(200,100,32);map_clear(0,graph,rgb(255,255,0));
x=200;
y=150;
idcanal=play_wav(idwav,0);
/*Las cinco líneas siguientes lo único que hacen es alterar el sonido emitido por
play_wav de manera que suene más alejado, con menor volumen y balanceado hacia la
izquierda. Serían opcionales, en todo caso*/
set_wav_volume(idwav,100);
set_channel_volume(idcanal,100);
set_panning(idcanal,100,0);
set_distance(idcanal,100);
set_position(idcanal,100,100);
while(is_playing_wav(idcanal))
  if(key(_s)) pause_wav(idcanal); end
  if(key(_r)) resume_wav(idcanal); end
          frame;
end
unload_wav(idwav);
end
```

Al volumen del canal en vez de un valor concreto se le podría poner una variable, de manera que si todos los sonidos tienen esa variable se podrá controlar el volumen general del juego alterando el valor de esa variable.Recuerda que también existe la función **set_wav_volume()** -no confundir con la anterior- que establece el volumen de ese sonido

en sí de forma permanente, no sólo cuando se reproduce. Hay que tener cuidado con la suma de los efectos de las funciones **set_channel_volume()** y **set_wav_volume()***:* a la hora de mezclar primero afecta el valor del sample y luego el del canal.

## *Trabajar con música

Los wavs están muy bien para efectos especiales cortos, como disparos, puertas, saltos y otro tipo de sonidos que no se reproducen durante mucho tiempo. Pero para temas de fondo o bandas sonoras no son adecuados, ya que ocupan mucho espacio. Para eso están las funciones **load_song(), play_song(), stop_song(), pause_song() ,resume_song() o unload_song()** que permiten usar o bien archivos MIDI o bien archivos OGG (y también archivos WAV otra vez).

En el Apéndice B se explica que los archivos MIDI (.mid) ocupan muy poco espacio porque sólo contienen una representación de las notas que componen la música, y es la tarjeta de sonido la que se encarga (con menor o mayor fidelidad según su calidad) de reproducir los instrumentos. Por lo tanto, es el formato ideal si quieres crear juegos que ocupen lo mínimo posible. Los archivos OGG en cambio, contiene una grabación digital de sonido comprimida al máximo, y están recomendados por su excelente calidad; si quieres usar música grabada en tu juego, es el formato adecuado.

Quisiera comentar la existencia de un formato de archivo de sonido del cual no hemos hablado hasta ahora, y que tampoco utilizaremos en este curso, pero que creo que como mínimo has de conocer, y que las funciones que estamos tratando son capaces también de reproducir. Se trata de los módulos.Son archivos con extensión .xm, o .it, o .s3m, o .mod, los cuales contienen una representación de las notas que componen la música (como los .mid) y, además, una digitalización de cada uno de los instrumentos usados (sonido digitalizado), así que es un formato intermedio a los midi y a los wav, tanto en calidad como en tamaño: suenan más realista que los .mid, pero ocupan más.

Todas las funciones comentadas en este apartado, si no se indica explícitamente lo contrario, están definidas en el módulo “mod_sound”.

**LOAD_SONG(“FICHERO”)**

Esta función recupera de disco un fichero de música, para reproducir posteriormente con **play_song()** . Actualmente se soportan los siguientes formatos: OGG Vorbis (.ogg) , MIDI (.mid), módulos (.xm,.it, .s3m, .mod) y WAV (.wav)

Esta función devuelve un número entero que identifica la música, o -1 si el fichero no está en un formato reconocido. Puedes usar el número devuelto para reproducir la música mediante la función **play_song()**

PARAMETROS: STRING FICHERO : Nombre del fichero

VALOR RETORNADO: INT : Identificador de la música

**PLAY_SONG (MUSICA, REPETICIONES)**

Esta función inicia la reproducción de un fichero de música recuperado con la función **load_song()** . Dependiendo del tipo de fichero, sólo es posible reproducir un fichero de música a la vez, por lo que cualquier música que se estuviese reproduciendo debe detenerse primero con **stop_song()**,, ya que a diferencia de **play_wav(), play_song()** no trabaja con canales, por lo que sólo se puede reproducir una música a la vez

El segundo parámetro indica el número de veces que debe repetirse la reproducción. Puede usarse 0 para no repetirla, o -1 para repetirla contínuamente hasta que se detenga con **stop_song()**

PARAMETROS:

INT MUSICA : Identificador de la música
INT REPETICIONES : Número de repeticiones

**STOP_SONG()**

Esta función detiene la reproducción de cualquier fichero de música en curso inmediatamente.. Para un mejor efecto, prueba a usar la función **fade_music_off()**

**UNLOAD_SONG (MUSICA)**

Esta función libera la memoria ocupada por una música cargada en memoria con **load_song()** . Antes es necesario, si se está reproduciendo, detener la reproducción de la música con **stop_song()**

PARAMETROS: INT MUSICA : Identificador de la música

**FADE_MUSIC_IN (MUSICA, REPETICIONES, MILISEGUNDOS)**

Esta función equivale a **play_song()** , excepto que la música no empieza a reproducirse a máximo volumen sino desde el silencio, y va subiendo de volumen gradualmente hasta que, pasado el tiempo indicado en el parámetro *milisegundos* , el volumen es completo.

El segundo parámetro indica el número de veces que debe repetirse la reproducción. Puede usarse 0 para no repetirla, o -1 para repetirla contínuamente hasta que se detenga con **stop_song()**

Esta función es especialmente útil para reproducir efectos de sonido contínuos de este tipo (como por ejemplo, sonidos ambientales) en lugar de música.

PARAMETROS:

INT MUSICA : Identificador de la música
INT REPETICIONES : Número de repeticiones (0=1vez; -1 infinitas veces)
INT MILISEGUNDOS : Tiempo para llegar al volumen completo en milisegundos

**FADE_MUSIC_OFF (MILISEGUNDOS)**

Esta función detiene la música que esté ejecutándose actualmente (que haya sido iniciada mediante la función **play_song()**) bajando gradualmente el volumen hasta que se apague. El parámetro *milisegundos* indica el tiempo aproximado en el que se desea que la música haya terminado.

Obviamente justo después de llamar a esta función, la música no ha acabado inmediatamente. Para saber cuándo es seguro usar **unload_song()** o reproducir otra música, es preciso utilizar **is_playing_song()** para comprobar si la música ya ha terminado de apagarse.

PARAMETROS: INT MILISEGUNDOS : Tiempo para llegar al final de la reproducción en milisegundos.

**IS_PLAYING_SONG()**

Esta función comprueba si cualquier música reproducida anteriormente mediante **play_song()** todavía está sonando, en cuyo caso devolverá 1. En caso contrario (la música ya ha terminado y no se reproduce en modo de repetición, o bien se ha detenido con funciones como **stop_song()** o **fade_music_off()** ), esta función devolverá 0.

VALOR RETORNADO: INT : Indicador de si la canción se está reproduciendo (1) o no (0)

**PAUSE_SONG()**

Esta función detiene temporalmente la reproducción de cualquier fichero de música en curso, hasta que se continúe más tarde mediante la función **resume_song(). E**sta función detiene la reproducción de música inmediatamente. Para un mejor efecto, prueba a usar la función **fade_music_off()**

**RESUME_SONG()**

Esta función continúa la reproducción de música desde el punto donde se detuvo anteriormente mediante la función **pause_song()**

| SET_SONG_VOLUME(VOLUMEN) |
|---|
| Esta función establece el nivel general de volumen para la música entre un nivel de 0 (silencio total) y 128 (volumen original del fichero).<br><br>PARÁMETROS : INT VOLUMEN : Volumen deseado (0-128) |

Un ejemplo de estas funciones de música podría ser el siguiente código. En él, si se pulsa la tecla "a" se crea un proceso, el cual mientras esté vivo mostrará por pantalla un rectángulo de color amarillo. Y este proceso estará vivo el tiempo que dure en sonar un archivo OGG (llamado "musica.ogg"). El usuario podrá pausar y reemprender la ejecución del sonido con la tecla "s" y "r" respectivamente. En cambio, si se pulsa la tecla "b", se creará otro proceso -"proceso2()", el cual mientras esté vivo mostrará por pantalla un rectángulo de color rosa.Y este proceso estará vivo el tiempo que dura en sonar -mediante un fade in- el mismo archivo ogg. El usuario podrá realizar un fade out si pulsa la tecla "f".

```
import "mod_key";
import "mod_text";
import "mod_sound";
import "mod_map";

global
   int idsong;
end

process main()
begin
idsong=load_song("musica.ogg");
while(!key(_esc))
      if (key(_a))miproceso(); end
      if (key(_b))miproceso2();end
     frame;
end
unload_song(idsong);
end

process miproceso()
begin
graph=new_map(100,50,32);
map_clear(0,graph,rgb(255,255,0));
x=100;
y=75;
play_song(idsong,0);
set_song_volume(100);
while(is_playing_song())
     if(key(_s)) pause_song(); end
     if(key(_r)) resume_song(); end
     frame;
end
end

process miproceso2()
begin
graph=new_map(100,50,32);
map_clear(0,graph,rgb(255,0,255));
x=200;
y=75;
fade_music_in(idsong,0,3);
while(is_playing_song())
      if(key(_f)) fade_music_off(3); end
     frame;
end
end
```

¿Has probado de ejecutar los dos procesos a la vez -apretando "a" y "b"? Verás, como era de esperar, que los dos procesos se ponen en marcha mostrándose los dos rectángulos, pero en cambio, la música no suena doblada,

sino que cada vez que se ejecuta uno de los procesos, se para la música que estaba sonando en aquel momento y empieza la música del nuevo proceso, así que sólo se escucha esa música una sola vez cada vez, sin ningún tipo de superposición. Esto es así porque, a diferencia de las funciones de la familia "load_wav()", "play_wav()", etc, las funciones de la familia "load_song()", "play_song()", etc no funcionan con canales, por lo que sólamente se puede escuchar a la vez un sonido cargado con **load_song()**. Si quieres, haz la prueba de ejecutar el mismo ejemplo anterior pero utilizando las funciones de la familia "load_wav()": si utilizar los canales de forma adecuada (un canal diferente para cada proceso), verás que sí que sonarán las diferentes músicas a la vez cuando los procesos estén en marcha.

## *Trabajar con ventanas

Bennu es capaz de manipular el comportamiento de la ventana del juego que está interpretando en ese momento, dentro del contexto del sistema operativo donde se ejecuta. Por ejemplo, es capaz de minimizar por código la ventana, moverla por la pantalla, cambiarla de tamaño (o obtener dicha información), etc. Para ello hace uso de una serie de funciones, todas ellas definidas en el módulo "mod_wm".

### MINIMIZE()

Esta función minimiza la ventana de juego y la sitúa en la barra del sistema

### MOVE_WINDOW(POSX, POSY)

Esta función sitúa la esquina superior izquierda de la ventana del juego en las coordenadas (POSX,POSY) de la pantalla, tomando como origen de éstas la esquina superior izquierda de la misma.

PARAMETROS:
INT POSX: Coordenada horizontal de la esquina superior izquierda de la ventana respecto a la de la pantalla.
INT POSY: Coordenada vertical de la esquina superior izquierda de la ventana respecto a la de la pantalla.

Un ejemplo de **minimize()** y **move_window()**:

```
import "mod_text";
import "mod_key";
import "mod_rand";
import "mod_wm";

process main()
begin
    write(0,0,0,0,"ESC para minimizar la ventana");
    while(!key(_esc))
        move_window(rand(0,640),rand(0,480));
        frame;  //Necesario
    end
    delete_text(0);
    while(key(_ESC)) frame; end //No pasa nada hasta que suelte otra vez la tecla ESC
    minimize();
    while(!key(_ESC)) frame; end
end
```

### GET_WINDOW_POS(&POSX,&POSY)

Esta función almacena en sus dos parámetros el valor actual de la posición de la esquina superior izquierda de la ventana del juego, teniendo como origen de coordenadas la esquina superior izquierda de la pantalla.

PARAMETROS:
POINTER POSX: Coordenada horizontal de la ventana del juego respecto el extremo izquierdo de la pantalla
POINTER POSY: Coordenada vertical de la ventana del juego respecto el extremo superior de la pantalla

### SET_WINDOW_POS(POSX,POSY)

Esta función establece una nueva coordenada para la esquina superior izquierda de la ventana del juego, teniendo como origen de coordenadas la esquina superior izquierda de la pantalla

PARAMETROS:
INT POSX: Coordenada horizontal de la ventana del juego respecto el extremo izquierdo de la pantalla
INT POSY: Coordenada vertical de la ventana del juego respecto el extremo superior de la pantalla

Un ejemplo de **get_window_pos()** y **set_window_pos()**:

```
import "mod_wm";
import "mod_text";
import "mod_key";

process main()
private
          int posx,posy;
          int a,b;
end
begin
          set_window_pos(0,0);
/*En los write_var se podría haber utilizado directamente posx y posy, pero usamos a y b
para poder mostrar el funcionamiento de get_window_pos*/
          write_var(0,0,0,0,a);
          write_var(0,0,10,0,b);
          repeat
                    if(key(_a))
                       posx++; posy++;
                              set_window_pos(posx,posy);
                              get_window_pos(&a,&b);
                    end
                    if(key(_s))
                       posx--; posy--;
                              set_window_pos(posx,posy);
                              get_window_pos(&a,&b);
                    end
                    frame;
          until(key(_esc))
end
```

**GET_WINDOW_SIZE(&VENTREALANCHO,&VENTREALALTO,&VENTANCHO,&VENTALTO)**

Esta función almacena en sus dos primeros parámetros el valor actual del tamaño real de la ventana (incluyendo marcos y la barra de título), y en sus dos últimos el tamaño de la ventana definido con **set_mode()**

PARAMETROS:
POINTER VENTREALANCHO : Ancho real de la ventana del juego, incluyendo marcos y barra de título
POINTER VENTREALALTO : Alto real de la ventana del juego, incluyendo marcos y barra de título
POINTER VENTANCHO : Ancho de la ventana definido con **set_mode()**
POINTER VENTALTO: Alto de la ventana definido con **set_mode()**

Un ejemplo de la función anterior:

```
import "mod_wm";
import "mod_text";
import "mod_key";
import "mod_video";

process main()
private
          int a,b,c,d;
end
begin
          write_var(0,0,0,0,a);
          write_var(0,0,10,0,b);
```

```
          write_var(0,0,20,0,c);
          write_var(0,0,30,0,d);
          repeat
                    if(key(_a)) set_mode(200,400); get_window_size(&a,&b,&c,&d); end
                    if(key(_s)) set_mode(500,50); get_window_size(&a,&b,&c,&d); end
                    if(key(_d)) set_mode(320,240); get_window_size(&a,&b,&c,&d); end
                    frame;
          until(key(_esc))
end
```

| GET_DESKTOP_SIZE(&ANCHO,&ALTO) |
|---|
| Esta función almacena en sus dos parámetros el valor actual de la resolución del escritorio (ancho y alto) donde se esté visualizando el programa Bennu en ese momento.<br><br>PARAMETROS:<br>POINTER ANCHO: Ancho del escritorio (en píxels)<br>POINTER ALTO: Alto del escritorio(en píxels) |

Un ejemplo donde se logra situar la ventana en el centro de la pantalla, mediante las funciones **get_desktop_size()** y **set_window_pos()**:

```
import "mod_key";
import "mod_wm";
import "mod_video";

process main()
private
     int pantalla_x = 320;  //modo de video  x
     int pantalla_y = 200;  //modo de video y
     int alto, ancho;       //para guardar resolución pantalla
end
begin
     set_mode(pantalla_x,pantalla_y);
     get_desktop_size(&ancho,&alto); //obtener resolución pantalla
     set_window_pos((ancho/2)-(pantalla_x/2),(alto/2)-(pantalla_y/2));//centro la ventana
     while (!(key(_esc)))
        frame;
     end
end
```

Un ejemplo donde se muestra la utilización de **get_window_pos()** y **set_window_pos()** para conseguir un efecto de temblor y rebote de la propia ventana del juego:

```
import "mod_rand";
import "mod_wm";
import "mod_video";
import "mod_map";
import "mod_text";
import "mod_key";
import "mod_proc";

#define MAX_PROC  2  //Número de bolas que aparecerán botando en pantalla
#define SCR_W   640  //Anchura de la ventana
#define SCR_H   480  //Altura de la ventana

global
    int count;
    int ball;
    int down, lateral;
    int wposx, wposy;
end

process main()
begin
    set_mode(SCR_W,SCR_H,32,MODE_WAITVSYNC);
```

```
    ball = new_map(30,30,32); map_clear(0,ball,rgb(255,0,0));

    write_var(0, 30, 10, 4, wposx);
    write_var(0, 30, 20, 4, wposy);

    for (count=0; count < MAX_PROC; count++) ball(); end

    while (!key(_ESC))
        get_window_pos(&wposx, &wposy);
        //Rebote inferior: muevo la ventana para abajo
        if (down>0)
             wposy=wposy-8;
            set_window_pos(wposx, wposy);
            down=down-8;
        end
        //Rebote lateral: muevo la ventana a la derecha ó a la izquierda
        if (lateral > 0)
            wposx=wposx-4;
            set_window_pos(wposx, wposy);
            lateral=lateral-4;
            if (lateral < 0) lateral = 0; end
        else if (lateral < 0)
                wposx=wposx+4;
                set_window_pos(wposx, wposy);
                lateral=lateral+4;
                if (lateral > 0) lateral = 0; end
        end
        frame;
    end
    end
    let_me_alone();
end


process ball()
private
    int velocity_x;
    int velocity_y;
    int initial_velocity_y;
end
begin
    x=0; y=(SCR_H-32)*10;
    resolution=10;
    velocity_x=rand(10,80);
    initial_velocity_y=rand(-80,-(SCR_H-32));
    velocity_y=initial_velocity_y;
    graph = ball;
    loop
        x=x+velocity_x;
        //Si choca con las paredes
        if (x<0 or x>SCR_W*10)
            bounce_x_window(velocity_x/2);
            velocity_x=-velocity_x;
        end
        y=y+velocity_y;
        //Si choca con el suelo
        if (-velocity_y<=initial_velocity_y)
            bounce_y_window(velocity_y/8);
            velocity_y=-velocity_y;
        else
            velocity_y=velocity_y+20;
        end
        frame;
    end
end

process bounce_x_window(int velocity)
begin
```

```
    while (velocity !=0)
        wposx=wposx+velocity;
        set_window_pos(wposx,wposy);
        lateral = lateral + velocity;
        velocity=velocity/2;
        frame;
    end
end

process bounce_y_window(int velocity)
begin
    while (velocity !=0)
        wposy=wposy+velocity;
        set_window_pos(wposx,wposy);
        down = down+velocity;
        velocity=velocity/4;
        frame;
    end
end
```

También es interesante conocer la existencia de la variable global predefinida ***FOCUS_STATUS***. Cuando el juego funciona en una ventana y ésta está en primer plano, el valor de esta variable pasa a valer 1 automáticamente. En cambio, cuando la aplicación está en segundo plano o minimizada, este valor se pone a 0.Para que un juego esté bien integrado en el sistema operativo, se recomienda consultar esta variable cada frame y poner el juego en pausa cuando no está a 1, especialmente para los juegos que disponen de la opción de funcionar en una ventana.

Por último comentar la existencia también de la variable global ***WINDOW_STATUS***. Esta variable se pone automáticamente a 0 cuando la ventana del programa está minimizada (igual que ***FOCUS_STATUS***), y a 1 cuando está visible (pero a diferencia de ***FOCUS_STATUS***, incluso aunque esté en segundo plano, tapada por otras ventanas). En juegos que se ejecutan a ventana completa, esta variable se pone a 1 cuando la aplicación está visible, y a 0 si el usuario pasa al escritorio (por ejemplo, pulsando ALT+TAB en Windows, o el equivalente en otro sistema operativo).

Veamos esto último con un ejemplo:

```
import "mod_wm";
import "mod_say";
import "mod_key";
import "mod_video";

process main()
begin
          set_fps(1,0);
          repeat
                    //Quítale el foco a la ventana del juego y vuélveselo a dar.
                    say("Focus_Status: " + focus_status);
                    //Minimiza la ventana y vuélvela a maximizar
                    say("Window_Status: " + window_status);
                    frame;
          until(key(_esc))
end
```

## *Trabajar con el sistema: Introducir parámetros a nuestro programa por intérprete de comandos . Órdenes “GetEnv()” y “Exec()”

Hasta ahora hemos ejecutado normalmente nuestros programas invocando al intérprete desde la consola así: *bgdi miPrograma.dcb*. Lo que no hemos visto hasta ahora es que a la hora de ejecutar nuestro programa por el intérprete de comandos, disponemos de una posibilidad muy interesante: pasar parámetros a tu programa en el momento de iniciar su ejecución. Estos parámetros iniciales pasados por consola podrán ser recogidos por el programa y usados en consecuencia. De esta manera, podremos hacer programas que no tengan que ir preguntando interactivamente datos

al usuario: si éste se los aporta en el mismo momento de lanzar la aplicación, ésta los recogerá inmediatamente y se podrá ejecutar sin interrupciones.

Si queremos hacer uso de esta posiblidad, para invocar nuestros programas tendremos que escribir algo por ejemplo como: *bgdi miPrograma.dcb ValorParam1 ValorParam2*, donde en este caso habríamos pasado dos valores "iniciales" que serían recogidos en nuestro programa para poder trabajar con ellos. Fíjate que los diferentes valores se separan por espacios en blanco.

Para poder recoger y utilizar estos valores dentro de nuestro código, Bennu dispone de dos variables globales predefinidas:

***ARGC :*** Esta variable entera contiene el número total de parámetros recibido por el programa, incluido el propio nombre del programa. Así, en el ejemplo anterior, valdría 3 (2 parámetros más el nombre del programa)

***ARGV:*** Este vector contiene en cada elemento el valor del correspondiente parámetro recibido por el programa, incluido el propio nombre de éste. Así, éste se correspondería a *argv[0]*, el valor del primer parámetro a *argv[1]*, etc. Bennu guarda internamente espacio suficiente para 32 parámetros, pero el número real de parámetros recibidos se obtiene a partir de ARGC.

Los parámetros pueden ser enteros, decimales, cadenas...Deberá ser el programador el que al recoger los datos asigne convenientemente el tipo de datos pertinente a cada parámetro recogido.

Veamos un ejemplo de uso.

```
Import "mod_text";
Import "mod_key";
Import "mod_proc";

process main()
private
      int i;
end
begin
write(0,300,200,4,argc);
if (argc > 1) //Si se han pasado parámetros a nuestro programa desde la consola...
    for(i=1;i<=argc;i++)
         write(0,100,10*i+10,4,argv[i]); //Escribo los valores pasados uno detrás de otro
    end
else //Si no...
     write(0,100,100,4,"¡No me has pasado nada!");
end
Loop
    if(key(_esc)) exit();end
    Frame;
End
End
```

Llamaremos "ejemplo.prg" al archivo que contiene el código anterior. Para probar este código, deberás ejecutarlo desde la consola. Si escribes por ejemplo simplemente *bgdi pepe.dcb* verás lo que ocurre. ¿Y qué pasa si escribes *bgdi pepe.dcb 1 hola que tal 354.6 -4.7* ?

También podríamos utilizar ***ARGC*** y ***ARGV*** sin necesidad de emplear la consola de comandos. Si seleccionas tu programa DCB dentro de tu gestor de ficheros preferido (Explorer en Windows, Nautilus en Gnome, Dolphin en KDE...) y seleccionas a más a más otros archivos (con la tecla MAYUS), y arrastras el DCB al archivo *bgd*i, el vector ***ARGV*** de llenará de las rutas absolutas de los archivos seleccionados. Lo puedes comprobar con el código anterior.

Aparte de obtener datos mediante parámetros del intérprete de comandos, nuestros programas tienen otra manera de recibir información proveniente del sistema operativo sobre el que está funcionando: realizar consultas sobre los valores actuales de las variables de entorno especificadas. Y esto se hace con la función **getenv()**, definida en el módulo “mod_sys”.

*Una variable de entorno es una variable que tiene definida el sistema operativo en sí (es decir, Windows o GNU/Linux, no ningún programa en particular), y que pueden contener información acerca del sistema, como el número de procesadores detectados, la ruta del sistema operativo, la ruta de las carpetas temporales o de los perfiles de usuario,etc Estas variables del sistema pueden ser accedidas desde cualquier programa de terceros, que es de hecho para lo que sirve la función* ***getenv()*** *de Bennu .*

*En Windows, puedes observar la lista de variables de entorno definidas actualmene y sus valores si vas a "Panel de Control->Rendimiento y mantenimiento->Sistema->Opciones avanzadas->Variables de entorno". Desde allí también podrás modificar los valores de las variables que desees, e incluso podrás crear o destruir variables. En la consola de comandos puedes hacer algo parecido con el comando "set". Ejemplos de variables de entorno típicas que ya están predefinidas de serie en un sistema Windows pueden ser PATH (que vale las rutas -separadas por ;- que el sistema incorpora dentro de la lista de rutas donde buscará los ejecutables del sistema), TEMP (que vale la ruta de la carpeta temporal del sistema),etc.*

*Si usas GNU/Linux, existe este mismo comando, "set", para gestionar las variables de entorno de este sistema,. En dicho sistema habrá un listado de variables de entorno predefinidas diferente del de Windows (de hecho, las variables de entorno ni tan siquiera suelen ser las mismas entre distribuciones de Linux diferentes) , pero algunas son muy típicas y existen incluso en Windows, como PATH ó HOME (que indica la ruta de la carpeta personal del usuario actual)*

**GETENV("NOMBREVARIABLE")**

Esta función, definida en el módulo "mod_sys", obtiene el valor actual de la variable de entorno del sistema que se le ha pasado como parámetro, y entre comillas.

PARAMETROS:
STRING NombreVariable : Nombre de la variable de entorno del sistema cuyo valor se desea conocer.

VALOR DE RETORNO:
STRING : Valor de la variable de entorno especificada

De todas maneras, si no sabes lo que es una variable de entorno, seguramente no necesitarás utilizar esta función.

Un ejemplo de su uso podría ser el siguiente código:

```
Import "mod_text";
Import "mod_key";
Import "mod_sys";

process main()
begin
  write(0,0,0,0, "El valor de la variable PATH es : ");
  write(0,0,10,0, getenv("PATH"));
  write(0,0,20,0, "El valor de la variable TEMP es : ");
  write(0,0,30,0, getenv("TEMP"));
  write(0,0,40,0, "El valor de la variable OS es: ");
  write(0,0,50,0, getenv("OS"));
  write(0,0,60,0, "El valor de la variable WINDIR -sólo existente en Windows- es : ");
  write(0,0,70,0, getenv("WINDIR"));
  while(!key(_esc)) frame; End;
end
```

Un ejemplo práctico donde se demuestra la utilidad de este comando lo podemos encontrar en el siguiente código. En él se genera, gracias a la función "crear_jerarquia()", una jerarquía de carpetas dentro del perfil del usuario actual (cuya ruta se ha obtenido gracias a la variable de entorno correspondiente) para guardar allí un archivo con posible contenido a preservar en nuestro juego (puntos, récords,etc). La función "crear_jerarquia()" devuelve 0 si ha conseguido crear los directorios y -1 si algo ha fallado.La gracia del ejemplo está en que el lugar donde se guarda dicho archivo será diferente según el usuario que tenga iniciada la sesión y esté ejecutando el programa, con lo que disponemos así de un sistema para guardar datos personales de cada jugador en distintas carpetas, las cuales se llamarán

igual pero estarán situadas cada una en el perfil de cada usuario por separado. Además, este programa funciona tanto en Windows como en Linux.

```
Import "mod_key";
Import "mod_sys";
Import "mod_dir";
Import "mod_file";
Import "mod_say";
Import "mod_regex";

process main()
Private
 string carpetaperfil = "";
 //Atención a las barras!!.Todas han de ser así (/) y hay que añadir barra al final
 string carpetadatos = "/carpeta/dondeseguardaran/losdatos/deljugador/";
 string archivodatos = "nombredelarchivoquecontienelosdatos.dat";
 int fd=0;
end
Begin
  /*OS_ID es una variable global de Bennu que no hemos visto hasta ahora. Indica el
sistema operativo en el que está funcionando el intérprete de Fénix  (es decir, el
sistema operativo del usuario). En este if estamos comprobando que  su valor sea igual a
la constante OS_WIN32, el cual denota toda la familia de Windows de 32 bits.*/
 if (OS_ID == OS_WIN32) //Si estamos en Windows...
 /*...generamos la ruta absoluta que crearemos posteriormente en disco, a partir de la
ruta que obtenemos del valor de la variable APPDATA, concatenándole la ruta dada por
carpetadatos. La variable APPDATA es una variable de entorno que ha de existir por
defecto siempre en cualquier Windows 2000/XP/Vista*/
 carpetaperfil = getenv("USERPROFILE") + carpetadatos;
 else  //Si estamos en otro sistema (Linux,MacOS,BSD)
/*...generamos la ruta absoluta que crearemos posteriormente en disco, a partir de la
ruta que obtenemos del valor de la variable HOME, concatenándole la ruta dada por
carpetadatos. La variable HOME es una variable de entorno que ha de existir por defecto
siempre en Linux*/
 carpetaperfil = getenv("HOME") + "." + carpetadatos;
end

 if(carpetaperfil == carpetadatos)//La variable USERPROFILE no está definida,por alguna razón
  carpetaperfil = cd();   // Lo intentamos a  la desesperada en el directorio actual
  say("USERPROFILE no definida");
 end

 if(crear_jerarquia(carpetaperfil) == -1) // La creación del directorio ha fallado
  say("Save directory error..");//Avisamos al usuario y salimos:no intentamos guardar el fichero
 else
  /*Ahora que sabemos que hemos creado bien la ruta o que ya existía ya podemos crear la
ruta entera al fichero de partida guardada*/
  archivodatos = carpetaperfil + archivodatos;
  // Y ya podemos trabajar con él normalemente
  fd = fopen(archivodatos, O_WRITE);
  if (fd!=0) say("Aquí es cuando tendría que escribir algo en el archivo");
  else say("Write Error");
  end
  fclose(fd);
  say("Write OK: " + archivodatos);
 End
 while(!key(_esc))  frame; End
end


function crear_jerarquia(string nuevo_directorio)
Private
string directorio_actual="";
string rutas_parciales[10]; // Sólo acepta la creación de un máximo de 10 directorios
int i=0;
int i_max=0;
End
Begin
```

```
directorio_actual = cd(); //Recuperamos el directorio actual de trabajo, para volver luego a él
if(chdir(nuevo_directorio) == 0) // El directorio ya existe!
 cd(directorio_actual);
 return 0;
end
i_max = split("[\/]", nuevo_directorio, &rutas_parciales, 10);
chdir("/");
while (i<i_max)
 while(rutas_parciales[ i ] == "") // Se salta partes en blanco
  if(i++ >=i_max)
   cd(directorio_actual);
   return 0;
  end
 end
 if(chdir(rutas_parciales[ i ]) == -1)
  if(mkdir(rutas_parciales[ i ]) == -1) // Error al intentar crear el directorio
   cd(directorio_actual);
   return -1;
  end
  chdir(rutas_parciales[ i ]);
 end
 i++;
end
chdir(directorio_actual);
return 0;
End
```

Como se puede comprobar en el código, se ha hecho uso de una variable global predefinida de Bennu, ***OS_ID***, la cual guarda un valor numérico que representa el sistema operativo sobre el cual se está ejecutando el intérprete *bgdi*. Algunos de sus valores posibles más interesantes son:

| | | |
|---|---|---|
| **OS_WIN32** | Equivale a 0 | Microsoft Windows |
| **OS_LINUX** | Equivale a 1 | Linux |
| **OS_MACOS** | Equivale a 3 | Mac Operating System |
| **OS_BSD** | Equivale a 6 | Berkeley Software Distribution |
| **OS_GP2X_WIZ** | Equivale a 7 | Consola GP2X Wiz |

Por otro lado, en el módulo "mod_sys" hay definida otra función si cabe más interesante y potente que **getenv()**. Se trata de **exec().**

**EXEC(ESPERA,"EJECUTABLE", EJECARGC, &EJECARGV)**

Esta función, definida en el módulo "mod_sys", ejecuta cualquier aplicación externa instalada en el sistema sobre el cual está funcionando el código Bennu. La ruta de dicho ejecutable -o simplemente su nombre, si su ruta está incluida en el PATH del sistema- será el segundo parámetro de esta función. El primer parámetro de la función indica el modo de ejecución del programa externo, y puede valer lo siguiente:

| | | |
|---|---|---|
| **_P _WAIT** | Equivale a 0 | Indica que la ejecución del código Bennu se parará en el punto donde se lanzó la aplicación externa hasta que ésta sea finalizada, momento en el que se reanudará la ejecución del código Bennu en la siguiente línea. |
| **_P _NOWAIT** | Equivale a 1 | Indica que el código Bennu continuará su ejecución mientras la ejecución de la aplicación externa se inicia de forma paralela. |

El tercer parámetro indica el número de parámetros que se le pasa a la aplicación externa (similar en este sentido al concepto de la variable ***ARGC***), y el cuarto parámetro indica la dirección de memoria donde se aloja un vector que contiene en cada elemento el valor del correspondiente parámetro recibido por la aplicación externa (similar en este sentido al concepto de la variable ***ARGV***). Si la aplicación externa no recibe ningún parámetro, el tercer parámetro valdrá 0 y el cuarto, NULL.

Hay que tener en cuenta que, aunque esta función funciona exactamente igual en Windows y GNU/Linux, su uso

puede romper la independencia de plataforma de nuestro código Bennu, ya que dependiendo de la aplicación externa que se quiera invocar, ésta solamente estará disponible para un determinado sistema. Además, Bennu no controla en ningún momento si la aplicación a ejecutar está instalada, esta comprovación deberá hacerla el programador: si no se encuentra en el sistema, **exec()** devolverá error (-1).

PARAMETROS:
INT ESPERA: Modo de ejecución de la aplicación externa (“modo espera” -0- ó “modo no espera” -1-)
STRING EJECUTABLE: Ruta de la aplicación externa a ejecutar
INT EJECARGC: Número de parámetros pasados a la aplicación externa
POINTER EJECARGV: Puntero al vector cuyos elementos son los valores de los parámetros pasados a la aplicación externa

Por ejemplo, el siguiente código (sólo para Windows, pero es fácilmente modificable para otros sistemas) lo que hace es abrir el fichero “file.txt” mediante el “Bloc de Notas”. Debemos saber que si ejecutamos en el intérprete de comandos la siguiente línea: *notepad.exe file.txt* estaríamos haciendo esto mismo: “file.txt” es el primer (y único) parámetro que la aplicación externa notepad.exe. De ahí el valor del tercer y cuarto parámetro. El hecho de que el primer parámetro sea el modo “no espera” implica que a la vez que se lanza el “Bloc de Notas”, el código Bennu seguirá ejecutándose; como resulta que ya no hay más líneas por leer, el código Bennu acabará su ejecución, pasando como “testigo” la ejecución de notepad.exe.

```
Import "mod_sys";

Process Main()
Private
    string arg;
End
Begin
    arg = "file.txt";
    exec(_P_NOWAIT,"notepad.exe",1,&arg);
End
```

Otro ejemplo donde se ve mejor la diferencia entre el modo “espera” y el “no espera” es el siguiente. En él se ejecutan cuatro veces seguidas el “Bloc de Notas” en Windows ó bien Konqueror en GNU/Linux (previa detección de en qué sistema operativo estamos) para posteriormente ejecutarse cuatro veces seguidas también la calculadora de Windows o bien la calculadora de KDE (según otra vez el sistema que se detecte). Pero como podrás comprobar si pruebas el ejemplo, en el primer bucle se ejecutan las aplicaciones en modo espera, con lo que hasta que no se termine el primer “Bloc de Notas” no se continúa la ejecución de nuestro programa Bennu, el cual conlleva continuar con la siguiente iteración del for y la consiguiente nueva ejecución del “Bloc de Notas”. Así, durante cuatro veces, ejecutaremos el “Bloc de Notas “ siempre y cuando se haya cerrado el anterior. Por el contrario, cuando lanzamos la calculadora, no hay modo de espera, con lo que las iteraciones del bucle for se van sucediendo paralelamente a las ejecuciones de las diferentes aplicaciones externas, con lo que las cuatro calculadoras aparecerán prácticamente seguidas.

```
import "mod_sys";
import "mod_say";

global
 int i;
 int ret;
end

process main()
begin
    ret = launch_pad();
    say (ret);
    ret = launch_calc();
    say (ret);
end

process launch_calc()
begin
    /* Lanzo 4 procesos sin espera */
    for (i=0; i<4; i++)
```

```
        if (os_id == 0)
            exec(_P_NOWAIT, "calc.exe", 0, NULL);
        else
            exec(_P_NOWAIT, "kcalc", 0, NULL);
        end
    end
    return 1;
end

process launch_pad()
begin
    /* Lanzo 4 procesos con espera */
    for (i=0; i<4; i++)
        if (os_id == 0)
            exec(_P_WAIT, "notepad.exe", 0, NULL);
        else
            exec(_P_WAIT, "konqueror", 0, NULL);
        end
    end
    return 2;
end
```

Incluso podríamos ejecutar desde un código PRG un fichero compilado DCB, de manera que podríamos separar un gran programa en pequeñas porciones compiladas. Por ejemplo, podríamos escribir esto:

```
import "mod_sys";
import "mod_say";

process main()
private
    int error;
end
begin
    error=exec(_P_NOWAIT, "bgdi.exe otro.dcb", 0, NULL);
    if(error==-1) say("ERROR");end
end
```

donde el código de “otro.prg” sería cualquier cosa, por ejemplo:

```
import "mod_say";

process main()
begin
    say("Me ejecuto");
end
```

## *Trabajar con caminos (o sobre cómo lograr que los procesos realicen trayectorias inteligentes)

Para hacer que el gráfico de un proceso se dirija hacia un punto determinado de la pantalla sorteando de forma completamente autónoma todos los obstáculos (muros, enemigos, etc) que se interpongan a lo largo de su camino, deberemos utilizar las funciones **path_find(), path_wall()** y **path_getxy()**, todas ellas definidas en el módulo “mod_path”.

| **PATH_FIND (LIBRERIA,GRAFICO,X1,Y1,X2,Y2,OPCIONES)** |
|---|
| A partir de un gráfico de **8 bits** (especificado con los parámetros librería y gráfico), esta función, definida en el módulo “mod_path”, indica si es posible (o no) encontrar algún camino que parta desde el punto en (x1,y1) y que concluya en el punto (x2,y2) evitando cualquier obstáculo que pueda encontrar a su paso. Si es posible, (es decir, si se encontró un camino) esta función devolverá 1 (CIERTO); si no lo es (es decir, si no es posible la comunicación entre los puntos de inicio y fin), devolverá 0. No se asegura que el camino encontrado sea óptimo al 100%, sin embargo se |

acercará razonablemente. El origen de coordenadas para los puntos de origen y final del camino se toma desde la esquina superior izquierda del gráfico de 8 bits.

El gráfico de 8 bits es normalmente una representación en miniatura del mapa o de la superficie del juego (en miniatura porque es tontería malgastar recursos del ordenador manipulando una representación al mismo tamaño que el mapa real) , donde cada punto del gráfico representa una casilla o un área que puede o no estar ocupada con un obstáculo. Cada punto del gráfico de 8 bits contiene un color de su paleta (la cual ya sabemos que al ser de 8 bits puede albergar hasta un total de 256 colores, numerados desde el color 0 al color 255). Estos colores sirven para indicar a **path_find()** si la casilla en cuestión puede traspasarse o no: si el color es el número 0 de la paleta, se puede traspasar; si el color es cualquier otro de la paleta (desde el 1 hasta el 255 indistintamente), esa casilla es un obstáculo. Es por ello que normalmente, el gráfico de 8 bits solamente consta de dos colores que marcan respectivamente las zonas transitables (color 0) y las que no lo son (otro color).

En el caso de que **path_find()** devuelva 1 (es decir, que indique que sí existe algún camino posible), para ir obteniendo sucesivamente los puntos intermedios de que consta dicho camino será preciso emplear la función **path_getxy()**.

Se puede añadir, opcionalmente, cierta flexibilidad en el momento de la definición del camino mediante la función **path_wall()**, de forma que algunos valores superiores a 0 dentro de la paleta del gráfico de 8 bits,hasta un valor máximo, indicado como parámetro, también indiquen igualmente caminos transpasables, aunque con un coste superior (indicado por el propio valor del punto: es decir, el color 1 será más fácil de traspasar que el color 2, y así). De esta manera, las rutinas elegirán a veces un recorrido más largo, pero que pase por puntos de menor coste. Esta forma de operar puede tener gran utilidad en juegos en los que el mapa contiene casillas donde el de movimiento difiere; empleando esta facilidad, es posible hacer que los procesos escojan preferentemente moverse a través de carreteras y caminos antes que bosque a través, excepto cuando no hay ninguna carretera propicia para hacer el trayecto.

Los valores del último parámetro de **path_find()** pueden ser las siguientes constantes predefinidas (o una suma de ellas):

| | | |
|---|---|---|
| | 0 | Búsqueda normal |
| **PF_NODIAG** | Equivale a 1 | No permite el movimiento en diagonal (si no se especifica,sí se permite por defecto) |
| **PF_REVERSE** | Equivale a 2 | Provoca que **path_getxy()** -invocada más adelante- devuelva los puntos en orden inverso |

PARAMETROS:

INT LIBRERIA: Id. de la librería FPG que contiene el gráfico especificado como 2º parámetro
INT GRAFICO: Id. del gráfico de 8 bits que contiene la representación de los obstáculos
INT X1: Coordenada horizontal del punto, dentro del gráfico, origen del camino
INT Y1: Coordenada vertical del punto, dentro del gráfico, origen del camino
INT X2: Coordenada horizontal del punto, dentro del gráfico, destino del camino
INT Y2: Coordenada vertical del punto, dentro del gráfico, destino del camino
INT OPCIONES: Opciones de movimiento

**PATH_GETXY (&POSX, &POSY)**

Esta función, definida en el módulo "mod_path", actualiza dos variables (cuyas direcciones de memoria se le pasan como parámetro) con el contenido del siguiente punto del camino, encontrado en su momento por **path_find()**. Se deberá invocar repetidamente a **path_getxy()** para ir obteniendo uno a uno los puntos (x,y) que conforman dicho camino. Esta función devolverá 0 si no quedan más puntos en el camino (es decir, si ya se ha llegado al final), y en ese caso, ya no modifica las variables. Lógicamente, la primera llamada actualizará las variables con los valores del punto de inicio del camino. El origen de coordenadas para los puntos del camino obtenidos se toma desde la esquina superior izquierda del gráfico de 8 bits.

El tipo POINTER se utiliza en este caso para lograr que una función pueda devolver más de un valor a la vez: **path_getxy()** devuelve dos valores que se guardan en sus respectivas variables (cuyos nombres precedidos de & hemos colocado como primer y segundo parámetro), y que son actualizados en cada llamamiento a **path_getxy()**.

Por ejemplo, el siguiente bucle movería el proceso actual (ya que modifica sus variables locales ***X*** e ***Y***) a lo largo del camino encontrado por **path_find(),** suponiendo que el mapa de búsqueda es de una cuarta parte del tamaño de la pantalla:

```
WHILE (path_getxy(&a, &b))
        x =a* 4;
        y =b* 4;
        frame;
END
```

PARAMETROS:

POINTER POSX: Dirección de memoria de la variable entera POSX, donde se almacenará la coordenada horizontal del siguiente punto del camino.

POINTER POSY:Dirección de memoria de la variable enetera POSY donde se almacenará la coordenada vertical del siguiente punto del camino.

**PATH_WALL (VALOR)**

Esta función, definida en el módulo "mod_path", modifica el valor de "pared" empleado por la función **path_find()**. Este valor de pared será un número que indica la posición mínima del color dentro de la paleta (perteneciente al gráfico de 8 bits que representa el camino) a partir del cual un punto del gráfico es intraspasable. Es decir, si por ejemplo el parámetro vale 5, todos los píxeles del gráfico del camino pintados con los colores que ocupan las posiciones 0,1,2,3 y 4 de la paleta serán traspasables, y el resto no.

El orden de los colores traspasables influye en la facilidad de "traspase". En el ejemplo anterior, el color 1 es el más fácil de traspasar porque no ofrece apenas resistencia para los móviles; en cambio, el color 4 será más dificultoso, y en principio no será la opción escogida para crear un camino, a no ser que no haya otra opción. Es decir, la posición de los colores en la paleta se interpreta como el "coste" de movimiento de un punto dado, hasta llegar a la posición marcada por el parámetro, la cual marca la barrera infranqueable. Más concretamente: cada posición en la paleta requiere el doble de pasos que la anterior, hasta llegar a la posición infranqueable (1ª posición, 1 paso necesario; 2ª posición, 2 pasos necesarios,etc)

Evidentemente, esta función se ha de invocar siempre antes de **path_find()**, ya que ésta utiliza el valor previamente definido por **path_wall()** para calcular el camino más adecuado. Si no se utiliza esta función, por defecto, el valor de la "pared" que utilizará **path_find()** está a 1, lo que significa que sólo los puntos a 0 serán traspasables por el camino a resolver. Por otro lado, si a la función se le pasara como parámetro un valor menor a 0, éste no será modificado.

PARAMETROS:

INT VALOR:Posición mínima dentro de la paleta del gráfico de 8 bits, para los colores intraspasables

Como ejemplo de su uso, aquí tienes el siguiente código. Para hacerlo funcionar necesitarás dos imágenes PNG llamadas "mapcamino.png" (de 200x150 píxeles y obligatoriamente de 8 bits) y "mapvisible.png" (de píxeles 400x300 –el doble que la anterior-, de 32 bits o los que tú quieras). El primer gráfico representará el mapa invisible que especificará por dónde podrá pasar el móvil y por dónde no (comúnmente llamado mapa de pathfinding) ,y el segundo será la imagen que se muestre en pantalla.

Un ejemplo de mapa de pathfinding podría ser éste:

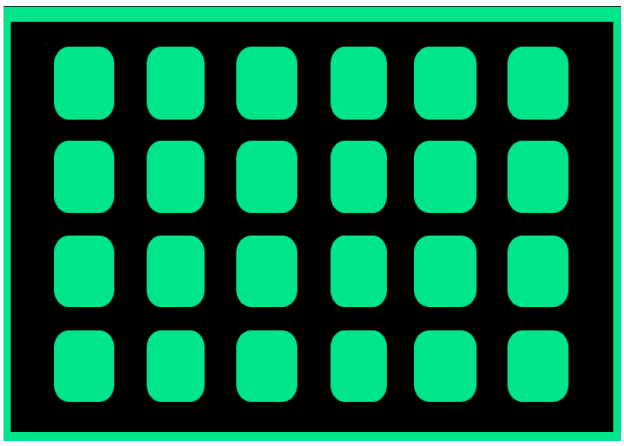
donde en color verde se han marcado las zonas libres de paso y en color amarillo los muros que no se pueden franquear, delimitando así los posibles caminos a seguir. Los colores concretos utilizados es lo de menos, lo importante es que, en este caso, el color verde ocupe la posición 0 de la paleta y el color amarillo cualquier otra (consulta el principio del capítulo 3 para recordar cómo se pueden crear imágenes de 8 bits con Gimp). En este caso, al haber sólo dos colores, tendremos un mapa de pathfinding sin "facilidades de traspasos" intermedios (es decir, no sería necesario utilizar **path_wall()** para nada), pero se puede utilizar cualquier otro mapa de pathfinding más sofisticado, con más colores. Evidentemente, el mapa visible ha de mostrar un decorado acorde a lo que el mapa de pathfinding establece.

```
Import "mod_video";
Import "mod_map";
Import "mod_screen";
Import "mod_path";
Import "mod_draw";
Import "mod_text";
Import "mod_key";
Import "mod_proc";

global
          int mapa; //Identificador del gráfico del mapa de pathfinding
          int i; /*Coordenada horizontal (dentro del mapa de pathfinding) de un punto
obtenido por path_getxy() que forma parte del camino*/
          int j; /*Coordenada vertical (dentro del mapa de pathfinding) de un punto
obtenido por path_getxy() que forma parte del camino*/
          int p; //Identificador de proceso. Usaremos su X e Y en el mapa visible
          int o; //Identificador de proceso. Usaremos su X e Y en el mapa visible
          int camino; //Lo que devuelve path_find() -1 si hay camino,0 si no-
end

process main()
begin
          set_mode(400,300,32);
          set_fps(40,0);

          //Cargamos el mapa de pathfinding...
          mapa = load_png("mapcamino.png");
          //...y mostramos el fondo
          put_screen(0,load_png("mapvisible.png"));

          //Éste sera el proceso que seguirá el camino...
          /*Sus coordenadas iniciales (los dos parámetros que tiene) se pueden
          cambiar para probar otras búsquedas de caminos) */
          p = prota(80,290);
          //...hacia este objetivo
          /*Sus coordenadas iniciales (los dos parámetros que tiene) se pueden
          cambiar para probar otras búsquedas de caminos) */
          o = objetivo(310,30);

          write(0,200,280,4,"Pulsa una tecla para buscar el camino.");
          //Mientras no pulse ninguna tecla el programa no hace nada
          while(scan_code==0) frame; end
          delete_text(0);
```

```
          /*Antes de calcular el camino, establecemos el color 4 como el
          mínimo número no traspasable de la paleta del mapa de pathfinding.
          Si éste tiene menos colores en su paleta, todos los colores serán
          traspasables, en menor o mayor medida.*/
          path_wall(4);
          /*Buscamos un camino, impidiendo movimientos en diagonal.
          Las coordenadas de los procesos las dividimos entre dos porque
          el mapa de caminos tiene la mitad de tamaño que el mapa visible*/
          camino = path_find(0,mapa,p.x/2,p.y/2,o.x/2,o.y/2,PF_NODIAG);

          if(camino == 0) //Si path_find devolvio 0, no hay camino posible
                    write(0,200,280,4,"No se encontró ningún camino válido.");
                    write(0,200,290,4,"Pulsa una tecla para salir...");
                    while(scan_code==0) frame; end
                    exit();
          else //Si se encuentra un camino...
                    write_var(0,0,0,0,i);/*...vamos mostrando las coordenadas de los
puntos que serán obtenidos sucesivamente por path_getxy()...*/
                    write_var(0,0,10,0,j);
                    while(path_getxy(&i,&j)) //...leemos el siguiente punto del  camino
                              p.x=i*2; //multiplicamos el punto por 2, ya que el mapa
                      //de pathfinding es la mitad de grande que el grafico del fondo.
                              p.y=j*2;
                              frame;
                    end
                    delete_text(0);
                    write(0,200,280,4,"El proceso ha llegado al destino.");
                    write(0,200,290,4,"Pulsa una tecla para salir...");
                    while(scan_code==0) frame; end
                    exit();
          end
end

process prota(x,y)
begin
          z=-1;
          graph = new_map(8,8,32);
          drawing_map(0,graph);
          drawing_color(rgb(255,255,0));
          draw_fcircle(3,3,3);
          loop
                    frame;
          end
end

process objetivo(x,y)
begin
          z=0;
          graph=new_map(16,16,32);
          drawing_map(0,graph);
          drawing_color(rgb(0,255,255));
          draw_box(0,0,15,15);
          loop
                    frame;
          end
end
```

Observa los comentarios incluidos en el código fuente:explican paso a paso todo el programa. Cuando lo hayas comprendido, puedes jugar probando variantes del mismo, como por ejemplo cambiar las posiciones iniciales de los procesos “prota()” y “objetivo()”, variar el mapa de pathfinding para incluir más obstáculos de diferente dificultad sin ningún camino libre, o bien incluir la opción de PF_REVERSE en **path_find()**, etc.

A continuación presento otro ejemplo, muy parecido al anterior. Primero hemos de disponer de un mapa de pathfinding en forma de retícula como el de la imagen inferior (y de su correspondiente imagen de fondo visible, en este caso del mismo tamaño), donde las zonas de color serán las zonas prohibidas. En este ejemplo sólo hay dos niveles de muro: 0 si es traspasable y 1 si no, no hay grados intermedio



La idea es la siguiente: colocaremos un proceso en algún lugar de esta retícula y con el ratón podremos ir haciendo clic en cualquier sitio. Si el clic lo hacemos en un punto al cual se puede llegar a través de alguno de los pasillos "negros" que forman la retículoa, el proceso se moverá hacia el punto donde hemos hecho clic. Si, en cambio, hacemos clic en el interior de alguna de las zonas de color (zonas prohibidas), el proceso no se podrá mover.*(NOTA: existe un bug en la función path_find() que hace que si no existe ningún camino posible, el programa se bloquee. Ojo. Habría que implementar de nuevo el algoritmo interno, ya sea un algoritmo de búsqueda A*, Dijkstra ó similar).*

Necesitarás un FPG llamado "path.fpg" con tres gráficos: el 001 será el gráfico del proceso que moveremos, el 002 será el fondo visible raticular, de 800x600, y el 003 será el mapa de pathfinding raticular, del mismo tamaño.

```
import "mod_text";
import "mod_map";
import "mod_mouse";
import "mod_proc";
import "mod_key";
import "mod_path";
import "mod_video";
import "mod_screen";
import "mod_math";
import "mod_draw";

global
          velocidad=1;
end

process main()
begin
//El modo de video no importa, pero la profundidad del fpg debe ser de 8bits
set_mode(800,600,32);
load_fpg("path.fpg");
put_screen(0,2);
pt();
//Muestro el puntero del ratón
mouse.graph=new_map(10,10,32);map_clear(0,mouse.graph,rgb(255,255,255));
loop
          if (key(_esc)) exit(); end
          if (key(_up)) velocidad++; end  //velocidad de desplazamiento
          if (key(_down)) velocidad--; end
          frame;
end
end


process pt()
private
          string cadena, cadena2;
end
begin
```

```
graph=1;
x=45; y=64; //deben estar en una zona negra de tu gráficos
write_var(0,x,y,4,cadena);
write_var(0,x,y+20,4,cadena2);
write_var(0,x,y+40,4,velocidad);
loop
/*Para que el proceso siga al ratón, éste ha de estar relativamente cerca (la función
path_find tiene un rango de acción limitado). Se indica cuándo se está y cuando no.*/
          if (fget_dist(mouse.x,mouse.y,x,y)>350)
                cadena="Muy lejos";
          else
                cadena="Vamos";
          end
//Si el ratón está sobre un punto imposible de llegar, se indica.Igual en caso contrario.
          if (map_get_pixel(0,3,mouse.x,mouse.y)<>0)
                cadena2="Imposible llegar";
          else
                cadena2="Vamos";
          end
          if (mouse.left!=0)
                    if (path_find(0,3,x,y,mouse.x,mouse.y,PF_NODIAG))
                            while (path_getxy(&x, &y))
//Este bucle for se utiliza para acelerar el movimiento del proceso.Se explica luego
                                  for (z=0;z<velocidad-1;z++);
                                          path_getxy(&x, &y);
                                  end
                                  frame;
                            end
                    end
          end
          frame;
end
end
```

En el código anterior se ha utilizado bucle FOR similar a éste:

```
for (z=0;z<velocidad;z++);
      //si se puede le pasa las nuevas coordenadas a las variables idx, idy
      path_getxy(&idx, &idy);
end
x=idx;  y=idy;   //lo hace avanzar a las nuevas coordenadas
frame;
```

¿Qué hace esto? Fíjate, se repite la ejecución de **path_getxy()** tantas veces como el valor que se le haya dado a *velocidad*. De esta manera, estaremos avanzando en cada iteración un punto más en el camino a seguir, pero como no hay ningún frame dentro del FOR, este avance es invisible por pantalla. Sólo cuando se sale del bucle, se asignan los valores finales dados por **path_getxy()** a la posición del proceso actual y se ejecuta la sentencia frame (¡sólo entonces!), es cuando se mostrará por pantalla el recorrido realizado por el proceso, el cual comprenderá varios puntos de golpe. Evidentemente, cuanto mayor sea el valor de *velocidad*, más iteraciones se realizarán y más puntos se recogerán en **path_getxy()** hasta el punto final y a la orden frame, por lo que la habrá una sensación de mayor velocidad en el movimiento del proceso.

Otro ejemplo práctico de pathfinding podría ser programar un circuito de carreras donde los coches sólo pueden seguir el trazado marcado porque si salen de la pista reducen drásticamente su velocidad. Aunque esto también se podría realizar de una forma más sencilla utilizando mapas de durezas y regulando la velocidad a partir de ellas (este tema lo estudiaremos en el capítulo siguiente)

## *Trabajar con el Modo7

El Modo7 es un tipo muy particular de scroll en el cual el plano de éste no es paralelo a la pantalla sino que está abatido, dando una sensación de profundidad que puede simular un efecto 3D. Una muy limitación importante que sufre es que todos los gráficos relacionados con el Modo7 (tantos los gráficos de fondo scrolleables como los

gráficos que se moverán sobre éstos) han de ser de 8 bits . Recuérdalo.

Empezaremos viendo un ejemplo para entender mejor el Modo7. Para ello, nos tendremos que crear unos nuevos gráficos de 8 bits (en el capítulo sobre gráficos se explica cómo hacerlo con el Gimp).En concreto,dos gráficos: uno llamado "fondo8bits.png" de 640x480 que representará el fondo de scroll, y otro llamado "nave8bits.png" de 30x30 que representará un objeto (una nave) colocada en un lugar concreto dentro de este scroll (también podríamos haber puesto ambos gráficos dentro de un FPG).La idea es que aparecerá un scroll abatido, en el cual nos podremos mover como si tuviéramos una cámara en primera persona. También aparecerá un proceso inmóvil en un sitio concreto dentro de ese scroll.

```
import "mod_m7";
import "mod_map";
import "mod_video";
import "mod_key";
import "mod_screen";
import "mod_grproc";
import "mod_proc";

process main()
private
        int fondo;
        int id_camara;
end
begin
          set_mode(640,480,8);
          fondo=load_png("fondo8bits.png");
//Definimos la región (centrada y más pequeña que la pantalla) sobre la que haremos el m7
          define_region(1,0,0,640,480);
//Este proceso lo situaremos dentro del scroll como un elemento más del paisaje
          nave();
//Este proceso es el que moveremos por el modo 7 y usaremos como visión
          id_camara=camara();
          /*Creamos el modo 7, con los parámetros:
          número       : 0 en nuestro caso. Puede haber hasta 10 scrolls Modo7 (de 0 a 9)
          fpg                : librería de la que cogemos el gráfico del scroll m7
          gráfico_interior  : número del gráfico para el fondo del scroll m7
          gráfico_exterior  : este gráfico bordeará al anterior, 0 para ninguno
          región            : región de pantalla donde se visualizará el m7, en nuestro
caso la 1, que hemos definido antes, un poco más pequeña que la pantalla
          horizonte         : altura del horizonte (en pixels)*/
          start_mode7(0,0,fondo,0,1,100);
          //La cámara del scroll Modo7 será el proceso "camara()"
          m7[0].camera=id_camara;
          loop
                    frame;
                    if(key(_esc)) break; end
          end
          let_me_alone();
end

process nave()
begin
/*La variable local ctype indica el sistema de coordenadas del proceso.En este caso, está
"dentro del scroll" de ahí que le pongamos c_scroll para indicarlo*/
          ctype=c_m7;
          graph=load_png("nave8bits.png");
          x=300; y=275;
          loop
                    frame;
          end
end

process camara()
begin
/*La variable local ctype indica el sistema de coordenadas del proceso.En este caso, está
"dentro del MODO7" de ahí que le pongamos c_m7 para indicarlo*/
          ctype=c_m7;
```

```
        x=150; y=100;
        loop
                if(key(_left))  angle=angle+5000; end
                if(key(_right)) angle=angle-5000; end
                if(key(_up))    advance(5); end
                if(key(_down))  advance(-3); end
                 frame;
        end
end
```

Podemos enumerar las diferencias que encontramos entre los scrolls ya conocidos y éste:

La función que inicializa el scroll pasa de ser **start_scroll()** a **start_mode7()**, la cual está definida en el módulo "mod_m7".

El significado de los parámetros de ésta última son los mismos que el de la primera, excepto el 4º y el 6º:

*En **start_mode7()**, el cuarto parámetro también es un código de gráfico, pero a diferencia de **start_scroll()**, este gráfico no sirve para dibujar un segundo plano del fondo sino que sirve para dibujar un borde exterior infinito al gráfico principal (dado por el tercer parámetro).Es decir, el 4º parámetro sirve para extender el scroll de forma indefinida en todas direcciones, en forma de mosaico infinito. Prueba, en el ejemplo anterior, de asignar el identificador del gráfico de fondo (o el del gráfico de la nave, da igual) a este parámetro y mira lo que pasa.

*En **start_mode7()**, el sexto parámetro sirve para definir la altura del horizonte, en píxeles.Es decir, indicará a cuántos puntos desde la parte superior de la ventana se quiere situar la línea del horizonte. Si la cámara se sitúa por encima del plano abatido, entonces no se mostrará nada por encima de la línea del horizonte (ese hueco se suele rellenar con otra ventana de scroll o de modo 7); en cambio si la cámara se sitúa por debajo del plano, entonces no se mostrará nada por debajo de la línea del horizonte. Según esté más o menos elevada esta línea, dará la sensación de que la cámara está mirando más "hacia el cielo" o más "hacia el suelo".

Las 10 estructuras posibles que almacenan la información de los 10 scrolls posibles, en vez de llamarse ***SCROLL[]*** se llaman ***M7[].***

En el ejemplo se puede ver que esta estructura M7 también tiene un campo **"camera"**, cuyo significado es idéntico al que ya conocíamos, pero que en el caso del Modo7 es imprescindible establecer.

Para hacer que un proceso determinado pertenezca a un scroll Modo7, utilizamos también la variable ***CTYPE*** al igual que los scrolls ya conocidos , pero en el caso le asignamos el valor **c_m7** (que equivale al valor numérico 2).

La variable ***CNUMBER*** tiene el mismo significado y valores posibles que siempre.

Campos específicos de la estructura ***M7[]*** -que no hemos visto en el ejemplo anterior- que no están presentes en la estructura SCROLL ya vista son:

***"height"**: Establece la altura vertical (en píxeles) del proceso que hace de cámara del scroll respecto el gráfico de fondo.
***"distance"**: Establece la distancia horizontal en píxeles de la cámara visual respecto el proceso que hace de cámara. Este proceso se situará siempre delante de la cámara real (a la distancia especificada), y será ésta la que será manejada por el usuario y seguirá los movimientos de ésta. Si sólo se quiere una cámara sin visualizar un proceso delante, ya que es obligatorio tener un proceso cámara, se puede hacer que su ***GRAPH*** valga 0 para que sea invisible.
***"camera.X":** Coordenada X del proceso cámara dentro del gráfico de fondo del scroll
***"camera.Y":** Coordenada Y del proceso cámara dentro del gráfico de fondo del scroll
***"focus":** Focal de la visión
***"color":** Número de color dentro de la paleta que será el color de fondo del Modo7 si no se define ningún gráfico como mosaico en el 5º parámetro de **start_mode7()**
***"horizon":** Su significado es el mismo que el 6º parámetro de **start_mode7()**, pero este campo nos permitirá, una vez generado el scroll, modificar la altura de la línea del horizonte allí donde nos convenga, dando una sensación de movimiento de cámara vertical.
***"flags"**: Su significado es el mismo que el campo **"flags1"** de **SCROLL[]**

Así mismo, campos que conocemos de ***SCROLL[]*** pero que no están presentes en M7[] son: **"x0", "x1", "y0", "y1", "ratio", "follow", "flags1", "flags2", "region1", "region2".**

El campo **"z"**, en cambio, está en los dos tipos de scroll con el mismo significado.

Al igual que ocurría con los scrolls "normales", existe una función que sirve para parar (y dejar de mostrar por pantalla) el scroll de Modo7. Esta función (definida en el módulo "mod_m7") es **stop_mode7()**. Tiene un sólo parámetro, que se corresponde con el número de scroll Modo7 iniciado por **start_mode7()**

Veamos otro ejemplo similar al anterior, pero donde tendremos nuestra nave enfrente de la cámara siguiendo todos nuestros movimientos, y donde repetiremos hasta el infinito nuestro gráfico de fondo.

```
import "mod_m7";
import "mod_map";
import "mod_video";
import "mod_key";
import "mod_screen";
import "mod_grproc";
import "mod_proc";
import "mod_rand";

global
   int idy;
   int idx;
end

process main()
private
   int fondo;
   int i;
end
begin
          fondo=load_png("fondo8bits.png");
          set_fps(30,2);
          set_mode(640,480,8);
          start_mode7(0,0,fondo,fondo,0,100);
          M7[0].CAMERA=id; //El programa principal hará de cámara
          M7[0].HEIGHT=25; //A cierta altura del fondo
          M7[0].DISTANCE=150; //A cierta distancia del fondo original
          M7[0].CAMERA.X=50; //En una coordenada X concreta dentro del fondo original
          M7[0].COLOR=85;
          navecamara(320,240);
/*Pongo otras 60 naves inmóviles en coordenadas al azar, incluyendo en el fondo repetido
en mosaico*/
          while (i<60)
                    navenormal(rand(0,500), rand(0,500));
                    i++;
          end
          loop
               if(key(_esc)) exit(); end
               if(key(_left)) angle=angle+5000;end  //Muevo la cámara a la izquierda
               if(key(_right)) angle=angle-5000;end
               if(key(_up)) advance(10);end
               if(key(_down)) advance(-4);end

//Si llego a una determinada coordenada Y o X, la cámara no puede avanzar más.
               if(y=>1000) y=999; END
               if(y=<20) y=21; END
               if(x=>1000) x=999; END
               if(x=<20) x=21; END

/*Estas dos líneas sirven para pasar la coordenada de la cámara a variables globales que
luego usaremos para establecer a su vez esas coordenadas a los  procesos que queremos que
la sigan*/
                idx=x;
                idy=y;
                frame;
         end
end
```

```
process navenormal(x,y)
begin
          ctype=C_M7;
          graph=load_png("nave8bits.png");
          size=200;
          loop
                    frame;
          end
end

process navecamara(x,y)
begin
          ctype=C_M7;
          graph=load_png("nave8bits.png");
          loop
//La posición de nuestra nave será la misma que la de la cámara.
                    x=idx;
                    y=idy;
                    frame;
          end
end
```

A partir del ejemplo anterior, por ejemplo, podríamos implementar un sistema que detectara colisiones entre los ejemplos “navecamara()” y “navenormal()”.

Si quieres poner un gráfico de fondo en la parte superior de la pantalla, a modo de “cielo”, lo puedes hacer con un simple **put_screen()** o un **start_scroll()** estándar. También podrías poner el gráfico de un proceso, pero entonces lo que tienes que hacer es procurar que el proceso que muestre ese gráfico no pertenezca al scroll Modo7 (es decir, que no tenga la línea *ctype=c_m7;*) y que tenga una Z superior a la del scroll Modo7, el valor por defecto de la cual es de 256.

Como último ejemplo, aquí presento un código que intenta simular la implementación de un scroll Modo7 “a pico y pala”, sin utilizar **start_mode()**, únicamente utilizando funciones básicas de distancias y dibujo de Bennu:

```
import "mod_m7";
import "mod_map";
import "mod_video";
import "mod_key";
import "mod_screen";
import "mod_grproc";
import "mod_proc";
import "mod_math";

global
   int fondo;
end

process main()
begin
 set_mode(800,600,8);
 fondo= load_png("fondo8bits.png");
 amosPaYa();
 loop
    if (key(_esc)) exit();end
    frame;
 end
end

process amosPaYa()
private
        int dist,dx,dy;
        int x7,y7,z7=50,a7;
        int i;
end
begin
```

```
loop
  if (key(_up))    y7=y7+3;   end
  if (key(_down))  y7=y7-3;   end
  if (key(_a))     z7=z7+1;   end
  if (key(_z))     z7=z7-1;   end
  if (key(_left))  a7=a7+3000;end
  if (key(_right)) a7=a7-3000;end
  dist=fget_dist(0,0,x7,y7);
  dx=get_distx(a7,dist);
  dy=get_disty(a7,dist);
  //aqui se pinta el grafico usando rotacion + zoom
  for (i=200;i<590;i=i+2);
       define_region(10,0,i,800,2);
  /*La línea clave está aquí: pinto el gráfico del fondo según los valores de las
coordenadas x7,y7,z7 y a7 en aquél momento*/
       xput(0,fondo,400+dx,500+dy,a7,(((i-100)*2)*z7)/100,0,10);
  end
  frame;
end
end
```

# CAPITULO 9

## Tutorial para un RPG elemental

Un RPG (Rol Playing Game) es un tipo de juego que parte de una localización concreta (un mundo más o menos fantástico) y de uno o varios personajes principales, -cada uno de ellos con cualidades concretas y diferentes (más fuerte, más ágil,etc),- que se encargarán de realizar diferentes misiones para lograr al fin encadenar una historia en la que se obtiene una recompensa final. Durante el camino se enfrentarán a enemigos y conocerán amigos, podrán incrementar sus cualidades o decrementarlas,utilizarán objetos que hayan recogido antes para realizar diferentes acciones, viajarán por el escenario explorando mundos nuevos,etc. Su nombre viene de que cada personaje en realidad es un Rol, un papel, diferenciado de los demás personajes, con sus propias características.

En este sencillo tutorial, lo que haremos será construir un personaje que se mueva por un escenario (un campo verde) en las cuatro direcciones cardinales. Dentro de él, situaremos dos casas. Al principio nuestro protagonista sólo podrá entrar en una de ellas, donde encontrará otro personaje con el que podrá conversar, y el cual (si se conversa adecuadamente) ofrecerá a nuestro protagonista una llave, que le servirá para poder entrar en la segunda casa.También se incluirán enemigos que puedan acabar con la vida de nuestro amigo. Y ya está.

### *La pantalla de inicio

La pantalla de inicio del juego consistirá en un menú donde el jugador podrá elegir entre diferentes opciones: comenzar un juego nuevo y salir del juego, básicamente.

La resolución que escogeremos para la ventana de nuestro juego va a ser 320x200 píxeles; la puedes cambiar como quieras, pero ten en cuenta que las dimensiones de algunas imágenes también las tendrás que cambiar en consecuencia.

Crea los siguientes gráficos: un cursor (para el ratón) apuntando hacia la derecha de 40x40, y un fondo para la pantalla de inicio de 320x200. Mételos en un gráfico FPG llamado “rpg.fpg”, con el código 005 y el 605 respectivamente.

Crea además dos tipos de fuente, con el “FNTEdit” ó similar. Una fuente para el título del juego llamada “titlescr.fnt” y otra para las palabras del menú inicial “Jugar” y “Salir” llamada “select.fnt”.

El cuerpo esquelético de nuestro programa será algo así:

```
import "mod_map";

Global
          Int file1;
          Int select_fnt;
          Int title_fnt;
          Int level=1;
End

process main()
Begin
       set_mode(320,200,32);
          set_fps(60,1);
          file1=load_fpg("rpg.fpg");
          select_fnt=load_fnt("select.fnt");
          title_fnt=load_fnt("titlescr.fnt");
          Loop
```

```
                    Switch(level)
                              Case 1:
                                        Loop
                                                  frame;
                                        End
                              End
                              Case 2:
                                        Loop
                                                  frame;
                                        End
                              End
                              Case 3:
                                        Loop
                                                  frame;
                                        End
                              End
                              Case 4:
                                        Loop
                                                  frame;
                                        End
                              End
                              Case 5:
                                        Loop
                                                  frame;
                                        End
                              End
                    End
                    frame;
          End
End
```

Este programa no visualiza nada pero carga los gráficos. Seguidamente chequea (reiteradamente porque hay un bucle infinito) el valor de la variable *level* mediante un switch para ver si se tiene que cambiar de nivel de juego, ya que esta variable especifica a qué nivel de ha de ir en cada momento –en seguida se explica qué es eso de los niveles-. Verás que por defecto, (al principio del juego), la variable *level* vale 1 y por tanto, al iniciar el juego entraremos en el nivel 1 del juego siempre. De momento, en ese nivel lo único que hay es un LOOP con sólo un Frame, y por tanto no se muestra nada.

En este tutorial, el nivel 1 será la pantalla inicial de selección de opciones, el nivel 2 será el mundo scrollable por donde se moverá nuestro protagonista, el nivel 3 y el 4 representarán el interior de sendas casas dentro de ese mundo y el nivel 5 será una pantalla de selección de objetos (ítems del juego). Es decir, los niveles se pueden entender como las diferentes pantallas que podemos visualizar en nuestro programa, entre las cuales deberemos poder irnos moviendo: si estamos en el exterior del mundo estaremos en el nivel 2 pero si entramos en una casa estaremos en el nivel 3, y al salir de ella volveremos al nivel 2; cuando queramos coger un ítem de nuestro mundo entraremos en el nivel 5 y cuando queramos salir volveremos al nivel 1 donde estará el menú con la opción de acabar el juego. Piensa que hasta ahora no hemos hablado de niveles porque los juegos que hemos hecho sólamente constaban de una pantalla. Por ejemplo, si hubiéramos hecho un mata-marcianos donde después de acabar con x enemigos pasáramos por ejemploa otra fase más difícil con otros enemigos y escenarios, necesitaríamos utilizar este sistema de niveles aquí descrito (aunque no es la única solución). Fíjate que a pesar de que en cada nivel existe un LOOP infinito (y por tanto, a priori nunca se cambiaría de nivel), verás que pondremos diversos break; que harán que ya no sean infinitos.

Bien. Ahora, en el nivel 1 (la pantalla de selección inicial) pondremos el fondo, el título del juego y las palabras “Jugar” y “Salir”. Modifica el Case 1 así (además de incluir los módulos pertinentes):

```
Case 1:
          put_screen(file1,605);
          write(title_fnt,150,50,4,”Pon el título de tu juego aquí");
     write(select_fnt,150,140,4,"Jugar");
          write(select_fnt,150,170,4,"Salir");
          Loop
                    frame;
          End
End
```

Fíjate que las órdenes **write()** no las he puesto dentro del LOOP, porque si no nos habría saltado el famoso error que ocurre cuando hay demasiados textos en pantalla.

Ok, ahora vamos a hacer el cursor de selección de las opciones del menú. Como es un gráfico, que además tendrá que controlar el jugador, crearemos un proceso para él. Lo primero será modificar el programa principal para que llame al proceso "cursor()". Tendremos que hacer dos modificaciones: crear una nueva variable global llamada *selection*, con lo que la lista de variables globales hasta ahora sería:

```
Global
          Int file1;
          Int select_fnt;
          Int title_fnt;
          Int level=1;
          Int selection;
End
```

y modificar de la siguiente manera el Case 1:

```
Case 1:
          put_screen(file1,605);
          write(title_fnt,150,50,4,"Pon el título de tu juego aquí");
      write(select_fnt,150,140,4,"Jugar");
          write(select_fnt,150,170,4,"Salir");

          cursor();

          Loop
                    if(selection==1) level=2; break; end;
                    if(selection==2) fade_off();exit("Gracias por jugar");end
                    frame;
          End

          fade_off();
          delete_text(0); /*Si no borramos los textos, saldrán en los otros niveles*/
          clear_screen(); /*Si no borramos antes el fondo, cuando queramos poner otra
                                                  imagen de fondo –en el nivel 2- nos va a
dar error*/
          let_me_alone();
          fade_on();
End
```

Lo que hemos hecho aquí es crear el cursor y meternos en el LOOP a la espera de si el valor de la variable global *selection* llega a modificarse –de hecho, lo modifica el propio proceso cursor- a 1 o a 2. Esta variable representa la selección que hemos hecho con el cursor: 1 si queremos jugar y 2 si queremos salir, pero eso lo veremos ahora mejor en el código del proceso "cursor()". Fíjate de momento que si elegimos jugar, ponemos el valor 2 a la variable *level* y salimos del Switch. Es decir, salimos del switch diciéndole que cuando volvamos a entrar en él –porque el Switch está dentro de un LOOP- queremos ir al Case 2 (variable *level*).

Si elegimos salir, hacemos *con* **fade_off()** un borrado progresivo de la pantalla: es un efecto estético muy eficaz; y luego acabamos la ejecución del juego mostrando un texto. Las líneas que hay después del LOOP/END, evidentemente sólo se ejecutarán si hemos elegido Jugar, porque con Salir directamente ya se acaba todo. Estas líneas lo que hacen es hacer un fade off también, como transición al segundo nivel, borrar todos los textos que hubiera en pantalla, quitar la imagen de fondo puesta con **put_screen()** mediante la función nueva **clear_screen()**, matar todos los procesos que hubiera corriendo excepto el actual (es decir, en este caso matar el proceso "cursor()" solamente porque no hemos creado ningun más todavía) y poner un fade on con la función **fade_on()** para acabar la transición hacie el segundo nivel.

Seguidamente, escribiremos el código del proceso "cursor()" al final del programa principal:

```
PROCESS cursor()
PRIVATE
    Int c;
```

```
END
BEGIN
    file=file1;
    graph=5;
    x=70;
    y=140;
    c=3; //Necesaria para el efecto de tembleque
    selection=0; //Ponemos un valor inicial neutral para "selection"

    WHILE (NOT key(_enter) AND selection==0)
        IF (key(_up))   y=140;  END
        IF (key(_down)) y=170;  END
           IF (key(_esc)) selection=2; END /*Sale fuera del bucle como si se hubiera
escogido la opción de salir del juego */
      //Efecto tembleque horizontal
        x=x+c;
        IF (x>=80) c=-3; END
        IF (x<=70) c=3;  END

        FRAME;
    END

    //Una vez que se ha apretado el enter,se comprueba dónde estaba el cursor
    SWITCH (y)
        CASE 140: //Coord Y donde hemos escrito la palabra "Jugar" con el write
            selection=1;
        END
        CASE 170: //Coord Y donde hemos escrito la palabra "Salir" con el write
            selection=2;
        END
    END
END
```

La verdad es que no deberías molestarte mucho en entender este código.Lo único que hace es que podamos mover el cursor arriba y abajo para señalar las dos opciones disponibles jugando con las coordenadas de las variables locales ***X*** e ***Y***, y que éste tenga un efecto de que parezca que tiembla horizontalmente. Cuando apretemos ENTER se parará y asignará el valor correspondiente de la opción elegida (1 o 2)a la variable global *selection*, que, como hemos visto, será usada en el Case 1 para saber, en el momento que cambie de valor, a qué nivel se va.

Ahora mismo tu programa debería ir al nivel 2 (nuestro mundo) cuando apretemos ENTER estando el cursor en la opción de Jugar, o debería de acabarse con un bonito fade off si se apreta ESC o se elige la opción Salir. Antes de testear tu programa, sin embargo, modifiquemos rápidamente el Case 2 para que cuando estés en el mundo puedas regresar a la pantalla de inicio apretando la tecla ESC:

```
Case 2:
          Loop
                    if(key(_esc))
                              //Generar retardo
                              while(key(_esc)) frame; end

                              //Salir
                              level=1;
                              break;
                    end
                    frame;
          End
          fade_off();
          delete_text(0);
          clear_screen();
          let_me_alone();
          fade_on();
End
```

Así, ahora, cuando estés en el mundo, si apretas ESC la computadora esperará un momento y entonces volverá a la pantalla de selección inicial. Pero, ¿por qué se pone ese WHILE dentro del IF que no hace nada? Pues para

que mientras tengas pulsada la tecla ESC vayan pasando fotogramas (idénticos entre ellos) indefinidamente, creando así un estado de "pausa" hasta que dejes de pulsar la tecla ESC. Éste es un truco que se hace para que tengas la oportunidad de dejar de apretar la tecla ESC antes de poder hacer cualquier otra cosa como moverte a otro nivel: la computadora espera a que dejes de pulsar para continuar trabajando. Si no, lo que pasaría es que la computadora volvería enseguida a la pantalla inicial y posiblemente continuarías teniendo apretada la tecla ESC (el ordenador es más rápido que tu dedo), con lo que como en el nivel 1 si apretas ESC sales del juego, de golpe habrías acabado la ejecución del programa sin poder siquiera pararte en la pantalla inicial de selección.

El mismo código que hay en el Case 2 escríbelo en el Case 3 y el Case 4, el interior de las dos casas, para volver a la pantalla inicial. Deja el Case 5 sin tocar, ya que es la ventana de selección de ítems del juego.

Y la pantalla de selección ya está lista.

## *Creando el mundo y nuestro protagonista moviéndose en él

Primero haremos los gráficos:

-Haz un gráfico de 3000x3000 píxeles que sea un gran cuadrado verde, con algunas manchas marrones que representen montañas, o manchas verde oscuro que sean bosque, o lo que se te ocurra. Se pretende que sea el campo por el que nuestro personaje caminará. Guarda este gráfico en el fichero FPG con código 300.

-Dibuja el protagonista, el chico/a que caminará por nuestro mundo. Este juego tendrá lo que se llama vista isométrica, así vamos a dibujar el personaje mirando hacia el norte, el sur y el este. No es necesario dibujarlo mirando al oeste porque haremos que el gráfico del este cambie de orientación, como en un espejo.Así que necesitarás tres gráficos para el chico. Asegúrate de que todos sean de 20x20 píxeles. Guarda los gráficos en el FPG así: el que mira al sur –hacia tí- con código 1, el que mira al norte con código 6 y el que mira al este –hacia la derecha- con código 10.

-Por cada dirección a la que mira nuestro personaje, necesitaremos crear tres dibujos similares para animar la caminata del protagonista en esa dirección cuando se mueva. Así que crea tres dibujos más por cada dirección ligeramente diferentes entre ellos, y guárdalos de tal manera que tengas cuatro imágenes mirando al sur con los códigos 1,2,3 y 4, cuatro imágenes mirando al norte con los códigos 6,7,8 y 9 y cuatro mirando al este con los códigos 10,11,12 y 13.

Ya tenemos los gráficos necesarios para este apartado.

Primero añadiremos dos variables más a la lista de variables globales del proceso principal, *MyChar_Position_X* y *MyChar_Position_Y*, y ya puestos las inicializaremos. Estas dos variables representarán la coordenada X e Y del personaje principal dentro del escenario:

```
Global
          Int file1;
          Int select_fnt;
          Int title_fnt;
          int level=1;
          int selection;
          int MyChar_Position_X=1516;
          int MyChar_Position_Y=1549;
End
```

Y modificaremos el Case 2 del proceso principal, añadiendo las líneas que aparecen en negrita:

```
Case 2:
          MyCharacter(MyChar_Position_X,MyChar_Position_Y);
          start_scroll(0,file1,300,0,0,0);
          Loop
```

```
                    if(key(_esc))
                              while(key(_esc)) frame; end
                              level=1;
                              break;
                    end
                    frame;
          End
          fade_off();
          delete_text(0);
          clear_screen();
          let_me_alone();
          fade_on();
End
```

Fíjate lo que hemos hecho. Hemos creado un proceso "MyCharacter()", que será nuestro protagonista, con dos parámetros, que son su X e Y inicial. Además, hemos iniciado un scroll, (aunque recuerda que al utilizar **start_scroll()** lo único que haces es decir que unas imágenes determinadas se usarán posteriormente para que hagan scroll, pero si no se hace nada más y sólo se escribe **start_scroll()**, no se moverá nada. De hecho, **start_scroll()** se puede usar como sustituto de la función **put_screen()** porque hasta que al scroll no se le dé ninguna consigna posteriormente, **start_scroll()** lo único que hará será poner la imagen que se le diga como parámetro de imagen de fondo).

Según los valores de los parámetros que hemos escrito en **start_scroll()**, éste será el scroll número 0; los gráficos del scroll se cogerán del archivo referenciado por *file1*; el gráfico en concreto del primer plano será el gráfico 300 (el mapa verde);como gráfico de segundo plano no habrá ninguno; el scroll existirá en una región que es la número 0 (es decir, la pantalla completa); y el número 0 como valor del último parámetro indica que el scroll no estará formado por ningún mosaico: es decir, que cuando se llegue a los límites del cuadrado verde no existirá nada más. Recuerda que si ese parámetro valiera 1, por ejemplo, indicaría que el scroll estaría formado por un mosaico horizontal de fondo: es decir, que cuando por la izquierda o la derecha el personaje llegara al límite de la imagen del cuadrado verde, éste se repetiría otra vez como en un mosaico. Puedes consultar el significado de los otros valores posibles en el apartado correspondiente del capítulo anterior de este manual.

¿Y por qué queremos crear un scroll? Porque la idea es que nuestro personaje se vaya moviendo por el escenario de tal manera que el mundo se mueva bajo sus pies según la dirección del movimiento del personaje. De hecho, la idea es que nuestro protagonista nunca deje de estar en el centro de la pantalla, y lo que se mueva es el escenario, dando el efecto de que viaje a través del mundo. O sea, queremos crear un scroll donde la cámara (a la que éste seguirá) será nuestro personaje.Así pues,creamos el siguiente proceso para nuestro personaje:

```
Process MyCharacter (x,y)
BEGIN
          ctype=c_scroll;
          scroll[0].camera=id;
          graph=1;
          angle=0;

          Loop
                    if (key(_right) and not key(_up) and not key(_down))
                              x=x+2;
                              if (graph<=10 or graph>=13) graph=10; end;
                              graph=graph+1;
                              flags=0;
                    end

                    if (key(_left) and not key(_up) and not key(_down))
                              x=x-2;
                              if (graph<=10 or graph>=13) graph=10; end;
                              graph=graph+1;
                              flags=1;
                    end

                    if (key(_up) and not key(_left) and not key(_right))
                              y=y-2;
                              if (graph<=6 or graph>=9) graph=6; end;
                              graph=graph+1;
```

```
                    end
                    if (key(_down) and not key(_left) and not key(_right))
                              y=y+2;
                              if (graph>=3); graph=1; end
                              graph=graph+1;
                    end

                    frame;
          End //Loop
End
```

Aquí hay muchas cosas que no hemos visto todavía, y hay que explicarlas con calma. Lo primero que vemos es la línea:

```
ctype=c_scroll;
```

que como sabes, indica cómo se tendrán que interpretar los valores de las coordenadas ***X*** e ***Y*** del gráfico del proceso en cuestión: si su valor es "c_scroll", en vez de ser la esquina superior izquierda de la ventana el origen de coordenadas a partir del cual se mide la ***X*** e ***Y*** del proceso , lo que se está diciendo al proceso "MyCharacter()" es que los valores de sus variables ***X*** e ***Y*** se van a tomar respecto la esquina superior izquierda del primer gráfico del scroll (el cuadrado verde). En otras palabras: los movimientos de "MyCharacter()" serán relativos respecto el scroll, no respecto la pantalla. Y eso, ¿por qué? Porque será mucho más cómodo para nosotros programar teniendo en cuenta la posición de nuestro personaje respecto el mapa que no respecto a la pantalla. Por tanto, es mucho más inteligente tomar como referencia un origen de coordenadas del propio mapa para saber en qué sitio está nuestro personaje en cada momento, ya que todos los objetos que pongamos en el mapa (casas,etc) estarán colocados en unas coordenadas concretas dentro de él, así que si tomamos las coordenadas de nuestro personaje referenciadas al mapa, será mucho más fácil comparar coordenadas entre esos objetos y el protagonista y ver si éste choca contra una casa o no, por ejemplo.

Hay que decir que esta línea NO es opcional, porque es necesaria ponerla para que la línea que viene a continuación pueda funcionar bien. Y la línea que viene a continuación es muy importante:

```
scroll[0].camera=id;
```

Ésta es la línea que hace moverse al scroll, recuerda. Lo que hace es relacionar el movimiento del scroll con el movimiento de nuestro personaje, tal como nosotros queríamos. El efecto que veremos será que nuestro personaje va a quedarse inmóvil en el centro de nuestro juego y lo que irá moviéndose es el suelo, persiguiendo al personaje como una lapa. El valor *ID* es el código identificador del proceso "MyCharacter()".¿Por qué? Porque recordad que *ID* es una variable local que contiene eso precisamente, el código identificador generado automáticamente por Bennu en la creación de ese proceso concreto.Así que lo que se está diciendo en el proceso "MyCharacter()" es que el scroll coja como proceso a "perseguir" a ese mismo, al proceso "MyCharacter()".

Posteriormente a estas dos líneas, lo más digno a comentar son los diferentes ifs del movimiento del personaje. Hay cuatro ifs para las cuatro teclas del cursor. En cada uno se cambia las coordenadas ***X*** o ***Y*** del personaje (recordemos que respecto el mapa), y además, a cada pulsación del cursor se establece un nuevo gráfico dentro del rango adecuado para que de la impresión de animación al caminar. Es decir, que si hemos hecho que los gráficos del personaje mirando al este tengan los códigos del 10 al 13, a cada pulsación de tecla el gráfico irá cambiando: 10,11,12,13…y cuando llegue al final –gracias al ese if de una línea que hay- se volverá otra vez al 10, y así.

¿Para qué sirven las líneas *flags=0;* del primer if y *flags=1;* del segundo?. Recuerda que si la variable predefinida local ***FLAGS*** vale 0, muestra el gráfico tal cual; y si vale 1 muestra el gráfico espejado horizontalmente. Es decir, si asignamos el valor de 1 a ***FLAGS*** lo que estamos haciendo es dibujar el gráfico del proceso "mirando hacia el lado contrario" del original, que es justo lo que queríamos para convertir nuestro personaje que originariamente mira al este en nuestro personaje mirando al oeste.

Y ya está. Deberías tener nuestro personaje moviéndose a través del cuadrado verde. Lo podrás ver mejor si añades al cuadrado verde manchas (montañas, ríos…).

## *El mapa de durezas

Un mapa de durezas es un dibujo especial que se superpone al dibujo del mapa (el cuadrado verde) – y por tanto, tiene las mismas dimensiones- pero que permanece invisible en pantalla. Este dibujo normalmente tiene dos colores: un color para casi todo el mapa y luego otro diferente que sirve para remarcar zonas especiales. Estas zonas especiales tendrán que coincidir con lugares concretos del mapa visible, como una casa, una montaña,etc; por lo que controlar las coordenadas de los objetos dentro del mapa es crucial. Y estas zonas especiales dentro del mapa de durezas sirven normalmente para indicar las zonas del mapa visible por donde el personaje no podrá caminar: emulan una barrera física. Quedaría muy feo dibujar una montaña y que nuestro personaje la traspasara como hace hasta ahora sin más. Por tanto, el mapa de durezas es un sistema para indicar que, por ejemplo, la montaña dibujada en el mapa visible corresponde a una zona especial en el mapa de durezas y por tanto es imposible atravesarla. La técnica para conseguir esto es sencilla: si en un momento determinado el ordenador detecta que el personaje tiene unas coordenadas X o Y que coinciden dentro del mapa de durezas con una zona especial (porque detecta que en ese sitio el mapa de durezas ha cambiado de color), entonces evita que dichas coordenadas X o Y varien, y lo que verá el jugador simplemente es que al intentar cruzar un río, por ejemplo, el protagonista no lo puede pasar. Seguramente lo entenderás mejor cuando hagamos el código.

Como de costumbre, primero haremos los gráficos.

-En el mapa verde, dibuja una casa en vista isométrica (es decir, mostrando la fachada y sin perspectiva). Asegúrate que la coordenada de la esquina inferior izquierda de la casa sea (1440,1540) y que la coordenada superior derecha sea (1550,1480). También deberías dibujarle una puerta a tu casa. Graba el nuevo mapa con la casa dentro del FPG, con el mismo código 300. Como orientación, tu casa debería tener esta pinta:

-Ahora abre el mapa con la casa con tu editor de imágenes favorito. Repinta de un único color muy concreto y específico (por ejemplo, rojo) la casa entera, sin ningún reborde, y deja la puerta fuera, sin "sobrepintar". El resto del mapa, si quieres, puedes hacer que sea de un mismo color uniforme sin nada más, negro por ejemplo. Tendría que quedarte, pues, un mapa enteramente negro (por decir un color) excepto un cuadrado rojo con una muesca, que sería la puerta. Guarda este nuevo mapa (que es el mapa de durezas), con el código 102.

Es muy importante la elección de un color fijo para el rellenado de la casa, ya que la idea del uso del mapa de durezas es comprobar en cada momento si las coordenadas de nuestro personaje concuerdan con las coordenadas donde en el mapa de durezas aparece ese color determinado, y reaccionar en consecuencia. Es decir, que haremos comparaciones de colores: "si en esta posición del mapa de durezas –donde está el personaje- está el color X, entonces quiere decir que se ha encontrado con la casa". Por lo tanto, hay que tener perfectamente claro y codificado cuál va a ser el color que especifique la presencia de los objetos en el mapa, por que si no las durezas no funcionarán. Entonces, surge el problema de elegir el color. El color concreto no importa, puedes poner el que tú quieras, pero tendrá que ser siempre ése.

La función clave que realizará el proceso de comprobación del cambio de color en el mapa de durezas es **map_get_pixel()**. Ya sabes que esta función lo que hace simplemente es "leer" el color que tiene un píxel concreto de una imagen en concreto. Cuando digo "leer" un color me refiero a que devolverá un código numérico que representará inequívocamente un color determinado. En el caso de gráficos de 32 bits, el valor devuelto es una codificación de las componentes del color (rojo, verde, azul) que depende de la tarjeta de vídeo y del modo gráfico, por tanto, ese código dependerá de cada ordenador donde se ejecute el juego (de hecho, el color podrá cambiar ligeramente de ordenador a ordenador). Así que en principio el valor devuelto por **map_get_pixel()** no será fijo y esto podría ser un gran problema, pero veremos ahora cómo se puede solucionar.

Un truco para que con gráficos de 32 bits se puedan hacer mapas de durezas sin problemas es el siguiente: crear un gráfico auxiliar que sólo contenga puntos con los colores de referencia que hayas usado en el mapa de durezas. Al iniciar el programa (o el juego o cuando haga falta) leer de ese mapa cada punto y guardar el valor en una variable, que luego se usará para la comparación. De esta forma en vez de usar un valor absoluto sacado de tu ordenador concreto con **map_get_pixel()** contra el mapa de durezas y compararlo a palo seco, se usaría el que se obtiene del ordenador en el que se está ejecutando el juego mediante el uso de **map_get_pixel()** contra el gráfico auxiliar, de forma que si el color no es exactamente el mismo dará igual porque el programa funcionará de todas formas. Es como tener una pequeña biblioteca de píxeles en posiciones conocidas con colores conocidos, para usar con todos los mapas y gráficos que necesites.

Este truco es la mejor manera de no tener problemas. De todas maneras, en este tutorial no nos complicaremos más y como tampoco tenemos tantos objetos en nuestro mapa, no usaremos este gráfico auxiliar. Lo que haremos simplemente es recoger el código de color que identifica un obstáculo directamente del mapa de durezas con **map_get_pixel()**, meterlo en una variable y luego, cuando nos interese, realizar la comparación de ese valor recogido con el valor que devolverá (otra vez) **map_get_pixel()** en la posición donde esté el personaje en ese momento. Cuando veamos el código lo entenderás mejor, seguramente.

Recuerda finalmente que esta función tiene cuatro parámetros: el código del archivo FPG cargado de donde se sacará el gráfico a usar, el código de ese gráfico dentro del FPG, la coordenada horizontal (X) del píxel del que se leerá su color y la coordenada vertical (Y) del mismo.Es importante recordar que los valores del 3r y 4º parámetro tienen como origen de coordenadas el propio gráfico, siendo su esquina superior izquierda el punto (0,0).

Bien, volvamos al juego. Una vez creados los gráficos, lo primero será añadir unas cuantas variables globales, todas enteras, al programa principal. A partir de ahora en el tutorial sólo se listarán los nombres de las nuevas variables que se añadan, que son:

*despega_x*
*despega_y*
*obstacle*
*colorObstaculo*

Seguidamente, modificaremos el Case 2 del programa principal de la siguiente manera (los cambios son los marcados en negrita):

```
Case 2:
   MyCharacter(MyChar_Position_X,MyChar_Position_Y);
   start_scroll(0,file1,300,0,0,1);
   colorObstaculo=map_get_pixel(file1,102,1500,1500);

   Loop
            IF(map_get_pixel(file1,102,Son.x+despega_x,Son.y+despega_y)==colorObstaculo)
                              obstacle=true;
            ELSE
             obstacle=false;
          END

            if(key(_esc))
                     while(key(_esc)) frame; end
                     level=1;
                     break;
            end
            frame;
   End
   fade_off();
   delete_text(0);
   clear_screen();
   let_me_alone();
   fade_on();
End
```

¿Qué hemos hecho aquí? Con el primer **map_get_pixel()** lo que hacemos es obtener un código de color (el cual almacenaremos en la variable global colorObstaculo), que es el código que tendrá el píxel (1500,1500) de nuestro mapa de durezas. Este punto corresponde a un punto cualquiera del interior del cuadrado –rojo, en principio-

que representa el obstáculo casa. Podríamos haber cogido cualquier otro punto que esté dentro del cuadrado rojo,porque lo que interesa es el color, que es el mismo en todos esos puntos.

Y la siguiente línea es la clave: comparamos el color de referencia obtenido con el anterior **map_get_pixel()** con el color del píxel que corresponde en ese momento al centro del gráfico del protagonista.¿Y eso cómo se sabe? De momento olvídate de las variables *despega_x* y *despega_y*: ¿qué es lo que tenemos como 3r y 4º parámetro del ***map_get_pixel()***? *Son.X* y *Son.Y*. Acuérdate que ***SON*** es una variable predefinida que referencia al último proceso hijo del proceso actual. Estamos en el proceso principal, y fíjate que tres líneas antes, dentro del Case 2, hemos creado el proceso "MyCharacter()". Es decir, que ***SON*** hará referencia a nuestro personaje. Y con *Son.X* y *Son.Y* estamos accediendo a las variables locales ***X*** e ***Y*** del proceso Son, es decir, del proceso "MyCharacter()".Por lo que este **map_get_pixel()** lo que hace es lo dicho: coger el color del píxel que es el centro del personaje.

Esto se podría haber hecho de muchas otras maneras diferentes. Creo que esta es la más fácil. También sería posible, en vez de utilizar ***SON***, haber hecho que en el momento de crear el personaje en el proceso principal,se recogiera el identificador de ese proceso en una variable, y utilizar esta variable en vez de Son en el **map_get_pixel()**. La ventaja que tiene esto es que esa variable siempre contendrá el identificador de nuestro personaje, porque ***SON*** sólo lo tendrá hasta que el proceso principal cree un nuevo hijo –que se supone que no lo hará, pero a lo mejor en nuevas versiones del juego…- con lo que si aparece un nuevo hijo, ***SON*** dejará de referirse al protagonista y no funcionará nada. En seguida explicaré para qué sirven las variables *despega_x* y *despega_y*.

Cuando el color coincida, quiere decir que el personaje se ha chocado contra la casa, y en ese momento, lo único que hacemos es poner a "true" la variable *obstacle*. La idea es que el valor de esta variable global se compruebe de forma contínua en el proceso "MyCharacter()", y en el momento que cambie a "true", el movimiento de nuestro protagonista quede interrumpido en la dirección adecuada, a pesar de pulsar las teclas del cursor.

Fijarse que el IF lo escribimos dentro un bucle infinito para que constantemente estemos recogiendo el nuevo valor del color presente por donde pisa nuestro protagonista, pero en cambio, el primer **map_get_pixel()** lo escribimos fuera porque es fácil ver que sólo necesitamos asignar una vez el valor que queremos a la variable *colorObstaculo*.

Bueno, pero esto todavía no está acabado. Ahora, para que funcione,tendremos que modificar el proceso "MyCharacter()" para que responda a los obstáculos que se encuentre (las novedades del código están en negrita):

```
Process MyCharacter (x,y)
BEGIN
          ctype=c_scroll;
          scroll[0].camera=id;
          graph=1;
          angle=0;

          Loop

             despega_x=0;
             despega_y=0;

             if(key(_right)) despega_x=2; end;
             if(key(_left)) despega_x=-2; end;
             if(key(_up)) despega_y=-2; end;
             if(key(_down)) despega_y=2; end;


             if (key(_right) and not key(_up) and not key(_down)and obstacle==false)
                              x=x+2;
                              if (graph<=10 or graph>=13) graph=10; end;
                              graph=graph+1;
                              flags=0;
             end

             if (key(_left) and not key(_up) and not key(_down) and obstacle==false)
                              x=x-2;
                              if (graph<=10 or graph>=13) graph=10; end;
                              graph=graph+1;
                              flags=1;
             end
```

```
            if (key(_up) and not key(_left) and not key(_right) and obstacle==false)
                                y=y-2;
                                if (graph<=6 or graph>=9) graph=6; end;
                                graph=graph+1;
            end

            if (key(_down) and not key(_left) and not key(_right) and obstacle==false)
                                y=y+2;
                                if (graph>=3); graph=1; end
                                graph=graph+1;
            end
            frame;
         End //Loop
End
```

Lo más importante y fácil de entender es la modificación que se ha hecho a los cuatro ifs del movimientos del personaje. Simplemente se dice que sólamente se moverá en la dirección especificada por el cursor si la variable *obstacle* es false. Si no, no. Es lo que he dicho antes. Si el personaje choca contra la casa, en el proceso principal se pone *obstacle* a true y por tanto, en el proceso personaje, aunque tecleemos el cursor, éste no se moverá.

¿Y para qué sirven las variables *despega_x* y *despega_y* que vuelven a aparecer aquí? Quítalas un momento del **map_get_pixel()** del proceso principal (dejando el *Son.x* y el *Son.y*). Ejecuta el programa. Verás que parece que funciona bien cuando el protagonista choca contra la casa, pero una vez chocado, nos es imposible moverlo de allí: se ha quedado pegado, incrustado. ¿Por qué? Porque, tal como lo hemos hecho, al chocar contra la casa *obstacle* pasa a valer "true", y por ello, los ifs del movimiento dejan de responder a las teclas del cursor, ya que *obstacle* no es false, y ya no podemos hacer nada. ¿Cómo podríamos hacer para que una vez que el personaje choque contra la casa, podamos seguir moviéndolo en las direcciones permitidas? Una alternativa sería esperar un plazo de tiempo después del choque y cambiar automáticamente el valor de *obstacle* de true a false otra vez como si no hubiera pasado nada. Pero la solución más fácil creo que es la que ha hecho aquí.

Recuerda que la comprobación de choque entre el personaje y la casa (el segundo **map_get_pixel()**) se realiza constantemente. Bueno, pues la idea es que en realidad la comprobación del choque no se haga utilizando el punto central del personaje sino otro punto del personaje diferente según las circunstancias. Si pulsamos el cursor derecho, ese punto estará 2 píxeles a la derecha del centro, si pulsamos el cursor izquierdo, ese punto estará 2 píxeles a la izquierda del centro, si pulsamos arriba ese punto estará 2 píxeles arriba del centro y si pulsamos abajo, 2 píxeles debajo del centro. ¿Y por qué? Piensa, es un sistema muy ingenioso. Supongamos que nos acercamos a la casa desde su izquierda, es decir, pulsando el cursor derecho. Cuando el punto que está a 2 píxeles a la derecha del centro del personaje choque con la casa, según el Case 2 *obstacle* valdrá true y por tanto, tal como está en el proceso "MyCharacter()" no nos podremos mover en ninguna dirección. Sin embargo, si pulsamos el cursor izquierdo, para "despegarnos" de la casa, vemos que la comprobación contínua de **map_get_pixel()** en el Case 2 ahora se realizará con el punto que está a 2 píxeles a la izquierda del centro. Es evidente que este nuevo punto no estará chocando a la casa, porque el que choca es el que está a la derecha del centro. Por lo tanto, en ese momento *obstacle* volverá a valer false, y podremos ejecutar los ifs del movimiento otra vez. Y este sistema es extrapolable a las demás direcciones.¿Y por qué 2 píxeles? Porque hemos hecho que el personaje se mueva de 2 píxeles en 2 píxeles (eso se puede cambiar, claro). Así, la detección del choque siempre se realizará siempre "un paso" (un movimiento) antes de efectuarlo realmente. No es la única solución, pero posiblemente sea la más sencilla de implementar.

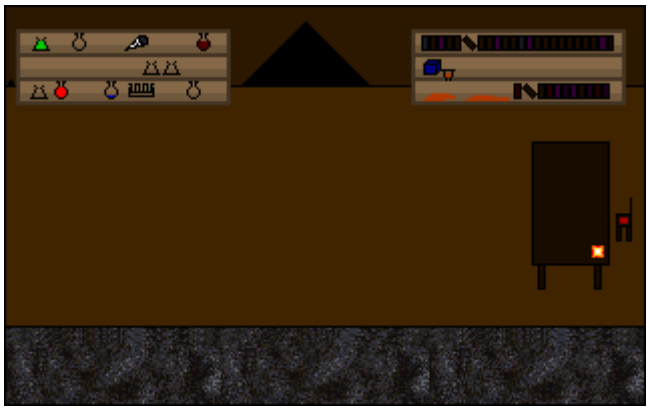
Y ya está. A partir de ahora simplemente tendrás que dibujar con el color que hayas elegido aquellas partes del mapa donde no quieras que el protagonista pueda ir. Chequea que tu programa funcione: en estos momentos el protagonista no podría atravesar la casa. Si ves que el protagonista la atraviesa sólo un poquito, tendrás que ajustar la ubicación de la mancha correspondiente en el mapa de durezas. Requiere cierta práctica con el tema de las coordenadas, pero al final lo harás cuidadosamente.

## *Entrar y salir de la casa

Primero, los gráficos:

-Dibuja un gráfico de 320x200 que represente el interior de la casa, como tú quieras.

Asegúrate de que la puerta esté en la parte superior de la imagen. Guardala dentro del FPG con código 401. Tiene que quedar algo similar –o mejor- a esto:



-Modifica tu mapa de durezas. Pinta ahora el hueco que dejaste sin pintar cuando rellenaste de rojo toda la casa (es decir, el hueco de la puerta de la casa) de otro color diferente al de la casa, por ejemplo, amarillo. Yo he dibujado la puerta de tal manera que el punto (1525,1535) está dentro de ella; si tú no lo tienes igual tendrás que cambiar estas coordenadas en el código que venga a continuación. Guárda el nuevo mapa de durezas otra vez dentro del FPG con el mismo código 102, claro.

Primero de todo,no te olvides de añadir dos variables globales enteras más a la lista: *house1* y *colorUmbral1*. E inicializa *house1* con valor 0.

Tal como comentá hace tiempo, el Case 3 del programa principal será el nivel del interior de esta casa (en seguida haremos una segunda casa, que será el Case 4). Por lo tanto, lo primero que hay que hacer es decirle al Case 2 cuando saldremos de él para ir al Case 3. En el Case 2 tendremos que añadir lo siguiente (las novedades están en negrita, y hay código que no se muestra porque es irrelevante: se ausencia se denota por puntos suspensivos):

```
Case 2:
   …
   colorUmbral1=map_get_pixel(file1,102,1525,1535);
   Loop
IF(map_get_pixel(file1,102,Son.x+despega_x,Son.y+despega_y)==colorObstaculo)          …
       ...
       END
IF (map_get_pixel(file1,102,Son.x+despega_x,Son.y+despega_y)==colorUmbral1)
           house1=true;
           END

      if(key(_esc))
                    …
      end
      if(house1==true)
                    house1=false;
                    level=3;
                    MyChar_Position_X=160;
                    MyChar_Position_Y=60;
                    break;
      end
      frame;
   End
 …
 //clear_screen();Con stop_scroll(0) ya no hace falta esta línea
 stop_scroll(0);
End
```

Fíjate que el proceso que hemos escrito ahora es muy similar al de antes: utilizamos una vez el **map_get_pixel()** para recoger en una variable (*colorUmbral1*) el valor numérico del color presente en el mapa de

durezas que representa la puerta, y una vez dentro del Loop, constantemente vigilaremos que nuestro personaje entre o no en contacto con dicha zona. En el momento que entra, fíjate que ponemos a 1 ("true") la variable *house1*, y esto lo único que implica es que al final de esa misma iteración se ejecutará el último if. Lo que hace este if es, primero, volver a poner *house1* a 0 –la "resetea"- para la siguiente vez que se ejecute el Case 2 se vuelva al valor inicial de esa variable, se especifica con *level* el nuevo nivel al que queremos ir, ponemos unas nuevas coordenadas para nuestro protagonista (ya que en el interior de la casa la puerta la he dibujado arriba a la izquierda de la pantalla, y por tanto el personaje, como representa que aparecerá por esa puerta, se visualizará en un lugar de la pantalla completamente diferente de donde estaba dentro del mapa verde), y finalmente salimos del bucle infinito, para poder escapar del Case 2.

Pero fíjate que hemos añadido, (justo antes de ejecutar otra vez en el Switch el cual volverá a hacer la comprobación de la variable *level* y entrará pues en el Case 3) una última orden muy importante: **stop_scroll()**, con la que detendremos el scroll del mapa verde,pues si no se parara, al entrar en el nivel 3 se seguiría viendo el mapa verde porque el scroll –y por tanto, su imagen asociada: el mundo exterior- se seguiría ejecutando.Se puede entender que **stop_scroll()** es una función equivalente a **clear_screen()**, sólo que la primera afecta a los fondos escrolleables que han sido creados mediante **start_scroll()**, y la segunda a los fondos estáticos creados con **put_screen()**

Antes de modificar el Case 3 para poder ver algo (si ejecutas ahora el programa verás que cuando entres en la casa no verás nada de nada porque no hemos ningún código visible en dicho Case), vamos a hacer otra cosa antes. Vamos a crear otro proceso llamado "MyCharacter_in_House()".¿Por qué? Porque el proceso que hemos creado para mostrar nuestro protagonista en el mundo verde ("MyCharacter()") no lo podemos utilizar para mostrarlo dentro de la casa debido a un pequeño detalle: la casa no tiene scrolling. Por eso tendremos que crear un proceso ligeramente diferente, y usarlo dentro de las casas y espacios sin scrolling.

Primero, añadiremos en el proceso principal dos nuevas variables globales enteras más, *increase_x* e *increase_y*, y luego escribimos esto:

```
Process MyCharacter_in_House(x,y)

BEGIN
      graph=1;
           angle=0;

           Loop

                      despega_x=0;
                      despega_y=0;

                      if(key(_right)) despega_x=2; end;
                      if(key(_left)) despega_x=-2; end;
                      if(key(_up)) despega_y=-2; end;
                      if(key(_down)) despega_y=2; end;

                      if (key(_right) and obstacle==false)
                                 x=x+2;
                                 if (graph<=10 or graph>=13) graph=10; end;
                                 graph=graph+1;
                                 flags=0;
                      end

                      if (key(_left) and obstacle==false)
                                 x=x-2;
                                 if (graph<=10 or graph>=13) graph=10; end;
                                 graph=graph+1;
                                 flags=1;
                      end

                      if (key(_up) and obstacle==false)
                                 y=y-2;
                                 if (graph<=6 or graph>=9) graph=6; end;
                                 graph=graph+1;
                      end

                      if (key(_down) and obstacle==false)
```

```
                                y=y+2;
                                if (graph>=3); graph=1; end
                                graph=graph+1;
                        end

                        frame;
                End  //Loop
End
```

Fíjate que este proceso es muy parecido a "MyCharacter()", pero ligeramente diferente. Básicamente, el cambio que hay es el haber eliminado las dos líneas donde se establecían los valores para ***CTYPE*** y *scroll[0].camera* respectivamente.

A más a más, vamos a crear un nuevo mapa de durezas para el interior de la casa. Abre el dibujo 401 y pinta el umbral de la puerta que dará otra vez al mundo exterior del mismo color que tenga la puerta de la casa en el mapa de durezas que hicimos antes para el mundo exterior –en nuestro caso,el color amarillo, que viene referenciado por nuestra variable *colorUmbral1-* , y vuelve a pintar del mismo color que tenga la casa en el mapa de durezas del mundo exterior,- en nuestro caso, el color rojo, que viene referenciado por nuestra variable *colorObstaculo-* todas aquellas partes de la casa donde no quieras que tu personaje se quiera mover. Fíjate que hemos seguido un criterio: en nuestro ejemplo las partes rojas de todos los mapas de durezas que hagamos serán zonas prohibidas y las zonas amarillas serán zonas de transición a otros niveles. Guarda este nuevo mapa de durezas en el FPG con el código 106.

Ahora sí que podemos modificar el Case 3 del proceso principal (las novedades están en negrita):

```
Case 3:
            put_screen(file1,401);
            MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);

            Loop
             IF (map_get_pixel(file1,106,son.x+despega_x, son.y+despega_y)==colorObstaculo)
                                obstacle=true;
                        ELSE
                                obstacle=false;
                        END

               IF (map_get_pixel(file1,106,son.x+despega_x,son.y+despega_y)==colorUmbral1)
                                house1=true;
                        ELSE
                            house1=false;
               END
               if(house1==true)
                    house1=false;
                    level=2;
                    MyChar_Position_x=1519;
                    MyChar_Position_y=1546;
                    break;
               end
                        if(key(_esc))
                                while(key(_esc)) frame; end
                                level=1;
                                break;
                        end
                        frame;
            End
            fade_off();
            delete_text(0);
            clear_screen();
            let_me_alone();
            fade_on();
End
```

Se parece bastante al Case 2:dibujamos un fondo, dibujamos el personaje,y a partir de entonces, comprobamos contínuamente que el personaje no choca con ninguna zona prohibida.(Si chocara, pondría la variable *obstacle* a true, con lo que automáticamente los ifs de movimiento del proceso "MyCharacter_in_House()" no se ejecutaría).De igual manera, se mantiene el mismo truco de *despega_x* y *despega_y*.Y también se comprueba que el personaje no choque con la zona del umbral de la puerta, en cuyo caso, sale del bucle y retorna al nivel 2. Fíjate que en el final no hemos puesto la función **stop_scroll()** porque no hay ningún scroll que parar, pero sí que hemos puesto la función **clear_screen()**, que borra del fondo la imagen anteriormente dibujada con **put_screen()**. Comprueba el funcionamiento de tu programa. Deberías de ser capaz de entrar y salir de la casa volviendo al mundo exterior.

## *Conversaciones entre personajes

Éste será posiblemente el apartado más duro de este tutorial, porque la clave de un buen RPG está en conseguir conversaciones interesantes de las que el personaje pueda sacar algún provecho para su aventura. Haremos dos nuevos personajes en este RPG, uno le dará al protagonista una espada después de que el protagonista haya hablado previamente con otro carácter y le haya dicho una respuesta correcta.

Primero, los gráficos:

-Crea tres dibujos de 20x20 de un carácter mirando al norte, este y sur (similar a nuestro protagonista, pero sin tener que preocuparnos de la animación de caminar). Grábalos dentro del FPG así: el gráfico que mira al sur con el código 900, el que mira al norte con código 901 y el que mira al este con código 902. Como anteriormente, no es necesario hacer un gráfico mirando al oeste porque podemos decir simplemente que la computadora haga el reflejo del gráfico que mira al este.Este dibujo representará un mago.

-Haz lo mismo para el otro personaje pero guarda sus dibujos así: el que mira al sur con el código 911, el que mira al norte con el código 912 y el que mira al este con el código 913.Este dibujo será Bob.

-Crea un dibujo de 20x20 de un cursor apuntando hacia la derecha, y grábalo con el código 14.

-Crea, finalmente, una nueva fuente con el FNTEdit, tan pequeña como sea posible (8x8 ya vale) y grábala como "talk_fon.fnt".

Bien. Primero crearemos los dos nuevos procesos que controlarán a los dos nuevos carácteres que acabamos de hacer. Primero, el mago:

```
Process Wizard(x,y)

Private
          Int id2;
          Int idangle;
          Int angletoChar;
End
Begin
          graph=900;
          id2=get_id(TYPE MyCharacter_in_House);

          Loop
                   idangle=get_angle(id2);
               angletoChar=idangle-90000;

               if(angletoChar>-45000 and angletoChar<45000)  graph=901;    end
               if(angletoChar>45000 and angletoChar>0)  graph=902;  flags=1;   end
               if(angletoChar<-45000 and angletoChar>-135000)  graph=902;  flags=0;  end
               if(angletoChar<-135000 and angletoChar>-195000) graph=900;   end
               frame;
     End
End
```

Y después, Bob:

```
Process Bob(x,y)

Private
          Int id2;
          Int idangle;
          Int angletoChar;
End
Begin
          graph=911;
          id2=get_id(TYPE MyCharacter_in_House);

          Loop
             idangle=get_angle(id2);
             angletoChar=idangle-90000;

             if(angletoChar>-45000 and angletoChar<45000)  graph=912;   end
             if(angletoChar>45000 and angletoChar>0)  graph=913; flags=1;   end
             if(angletoChar<-45000 and angletoChar>-135000) graph=913;  flags=0;   end
             if(angletoChar<-135000 and angletoChar>-195000)  graph=911;   end
             frame;
       End
End
```

En ambos procesos, el LOOP, con sus Ifs internos, para elegir qué gráfico en concreto se mostrará para ese nuevo personaje, dependiendo de la orientación que tengan respecto nuestro protagonista, de tal manera que siempre estarán de frente a él. Es decir, que cuando nuestro protagonista se mueva, tanto el mago como Bob corregirán su orientación para estar permanentemente mirándonos de frente.La clave está en dos funciones ya conocidas: **get_id()** y **get_angle()**.

Recuerda que **get_id()** devuelve el identificador del primer proceso que encuentre que se corresponda con tipo especificado. Si no encontrase ninguno, esta función devolverá 0. En las siguientes llamadas a esta función que se realicen utilizando el mismo parámetro, **get_id()** devolverá sucesivamente los identificadores de los siguientes procesos de ese tipo que se encuentren en memoria,hasta que ya no quede ninguno, en cuyo caso devolverá 0. También sería posible llamar a **get_id()** con un parámetro de tipo de proceso 0. En este caso, **get_id()** devolverá los identificadores de todos los procesos en memoria, de cualquier tipo, siempre que no estén muertos. Esta función es muy práctica si a la hora de crear los procesos no recogimos en variables los identificadores que devolvían.

Y ya sabes que ***get_angle()*** devuelve el ángulo formado por la línea que parte del centro del proceso actual y pasa por el centro del proceso cuyo identificador se especifica como parámetro, y la línea horizontal. El uso habitual de esta función consiste en hacer que un proceso"apunte" en dirección a otro: es el caso típico de monstruos y otrosobjetos que buscan siempre atrapar al jugador. Para que esta función devuelva un resultado útil, asegurarse de que lascoordenadas de ambos representan la misma zona (por ejemplo, que no estén cada uno en un área de scroll diferente).

¿Para qué usamos estas funciones en el código anterior? Lo que primero hacen es conseguir el identificador del proceso de tipo “MyCharacter_in_House()” (como sólo hay uno, no habrá confusión posible) y guardarlo en la variable privada *id2*.Y con este identificador en el bolsillo, en cada iteración del Loop se comprobará el ángulo formado por la línea horizontal y la línea que en ese momento vaya del centro del gráfico del proceso mago/Bob al centro del gráfico del proceso “MyCharacter_in_House()”. Después se hace una pequeña corrección, y el ángulo resultante será el que decida qué gráfico del mago/Bob (sur,norte,este,oeste) será el que se visualice en ese momento.

Bueno. Ahora nos falta incluir estos dos nuevos procesos en el interior de la casa. Por lo tanto, en el Case 3, JUSTO ANTES de la creación del proceso “MyCharacter_in_House()”, creamos los dos procesos:

```
Wizard(250,105);
Bob(130,100);
```

¿Por qué justo antes? Porque si te acuerdas, con las funciones ***map_get_pixel()*** utilizamos la variable ***SON***, que guarda el identificador del último proceso hijo del proceso actual. Como ese proceso hijo nos interesa que sea el “MyCharacter_in_House()”, debemos dejar este proceso como el creado inmediatamente anterior a la aparición de

**map_get_pixel()**.

Bien. Cuando ejecutes el programa y entres en la casa deberías ver los dos nuevos personajes en ella y éstos permanecerán siempre de frente mirando a la dirección donde esté el protagonista.Pero no.¿Por qué?Porque te he enredado. Si después de la línea **get_id()** en los procesos mago/Bob pones un **write()** para ver el contenido de *id2*, vale 0, y eso quiere decir que no se pilla bien el identificador de "MyCharacter_in_House()", y por tanto, ya no funciona todo lo que sigue. Y no pilla bien el identificador de "MyCharacter_in_House()" porque tal como lo hemos hecho, en el Case 3 hemos creado el proceso mago/Bob antes de "MyCharacter_in_House()", y por tanto, cuando el mago/Bob comienza a ejecutarse encuentra un proceso que todavía no existe. Por tanto, la solución será poner en el Case 3 la creación del mago/Bob después de la de "MyCharacter_in_House()", cosa que nos obligará, en el Case 3, a cambiar las líneas donde aparecía la variable ***SON*** (como parámetro de los **map_get_pixel()**) por otra cosa. Esto es relativamente fácil. Lo único que tenemos que hacer es almacenar en una variable global el identificador del proceso "MyCharacter_in_House()" cuando éste se crea, y utilizar esta variable global en vez de ***SON***. Es más, de esta manera incluso no sería necesario utilizar la función **get_id()** en los procesos mago/Bob porque ya tendríamos esa variable global que nos serviría. Por tanto, crea una nueva varible global llamada *idCharHouse* y haz todos estos cambios que te he comentado. Y por fin, funcionará.

Aprovecharemos este razonamiento que hemos hecho ahora y nos curaremos en salud. Cuando se crea el proceso "MyCharacter()" en el Case 2 haremos también que se recoja su identificador en una variable global, *idChar*, de tal manera que ahí también nos olvidemos de utilizar la variable ***SON***. En estos momentos no sería imprescindible hacer esto, pero por si las moscas, hazlo ya.

Lo que falta es decirle a la computadora qué es lo que ha de pasar cuando nuestro protagonista se acerque a los demás personajes. Seguiremos el mismo sistema de siempre: modificaremos el mapa de durezas de la casa para identificar dos nuevas zonas, que serán el radio de acción de los dos personajes inmóviles. Cada una de estas zonas las pintaremos de un color diferentes a los ya existentes: propongo azul para el mago y verde para Bob.

Y , previa inclusión de cuatro nuevas variables globales enteras, *colorMago, colorBob, near_wizard* y *near_bob*, añadiremos las siguientes líneas en el Case 3:

```
Case 3:
            …
            //Recogemos en colorMago el color de la zona del mago
            colorMago= map_get_pixel(file1,106,250,105);
            //Recogemos en colorBob el color de la zona de Bob
            colorBob= map_get_pixel(file1,106,130,100);
            Loop
            …
IF(map_get_pixel(file1,106, idCharHouse.x+despega_x, idCharHouse.y+despega_y)==colorMago)
                                    near_wizard=true;
                        ELSE
                                    near_wizard =false;
                        END

IF(map_get_pixel(file1,106, idCharHouse.x+despega_x, idCharHouse.y+despega_y)==colorBob)
                                    near_bob=true;
                        ELSE
                                    near_bob=false;
                        END
            …
            End
…
End
```

En seguida veremos qué consecuencias tiene el que *near_wizard* o *near_bob* valgan true o false.

Ahora lo que haremos será crear un nuevo proceso que pueda manejar el texto que aparecerá en pantalla cuando uno de los personajes hable. Primero, añade a la lista las variables globales enteras : *talk_font, talking, yes, no, MyChar_has_KeyOne, talkboblines, talkwizardlines* y *asked_bob_for_key* , y luego crea el siguiente proceso:

```
Process TalkLines()
Begin
```

```
          Loop
if(key(_space) and near_wizard==true and asked_bob_for_key==false and talkwizardlines==1)
                              talking=true;
                              Write(talk_font,160,180,4,"¿Qué quieres? Fuera de aquí");
                              //Retardo
                              frame(5000);
                              talking=false;
                              delete_text(0);
                    end


if(key(_space) and near_wizard==true and asked_bob_for_key==true and talkwizardlines==1)
                              talking=true;
                              Write(talk_font,160,180,4,"Ah, Bob dice que quieres la
llave. Bien, aquí la tienes");
                              Frame(5000);
                              talking=false;
                              delete_text(0);
                              MyChar_has_KeyOne=true;
                              talkwizardlines=2;
                    end

if(key(_space) and near_wizard==true and asked_bob_for_key==true and talkwizardlines==2)
                              talking=true;
                              Write(talk_font,160,180,4,"Ya tienes la llave, así que
fuera");
                              Frame(5000);
                              talking=false;
                              delete_text(0);
                    end

if(key(_space) and near_bob==true and asked_bob_for_key==false and talkboblines==1)
                              talking=true;
                              Write(talk_font,160,150,4,"¿Quieres la llave?");
                              Frame(5000);
                              Write(talk_font,150,170,4,"Si");
                              Write(talk_font,150,190,4,"No");
                              talk_cursor();
                              repeat
                                        frame;
                              until(yes==true or no==true)
                              talking=false;
                              delete_text(0);
                    end

                    if(near_bob==true and yes==true)
                              talking=true;
                              Write(talk_font,160,180,4,"Pregunta eso al mago");
                              Frame(5000);
                              talking=false;
                              delete_text(0);
                              asked_bob_for_key=true;
                              talkboblines=3;
                              yes=false;
                    end

                    if(near_bob==true and no==true)
                              talking=true;
                              Write(talk_font,160,180,4,"Bueno, está bien…");
                              Frame(5000);
                              talking=false;
                              delete_text(0);
                              talkboblines=1;
                              no=false;
                    end
```

```
if(key(_space) and near_bob==true and asked_bob_for_key==true and talkboblines==3)
                                talking=true;
                                Write(talk_font,160,180,4,"No hay nada más que me hayan
programado para decir…");
                                Frame(5000);
                                talking=false;
                                delete_text(0);
                    end

                    frame;
          End //Loop
End
```

Ya ves que este proceso lo que hace básicamente es comprobar si se cumplen unas determinadas condiciones, dependiendo de las cuales el mago y Bob dirán una frase u otra, e incluso tendremos la opción de responder nosotros a una pregunta que nos hagan. La idea es que primero se hable con Bob y luego el mago nos dé una llave. Si se va al mago sin haber hablado con Bob y haberle respondido correctamente, no nos dará nada. El primer if, por ejemplo, comprueba si se ha pulsado la tecla SPACE (es el truco para que aparezca el texto que dice el mago: si quieres esta condición se puede quitar y así el texto aparecerá automáticamente cuando nos acerquemos a él), comprueba si efectivamente estamos cerca de él,si ya hemos hablado con Bob para poder conseguir la llave (en este caso, no todavía) y si es la primera vez que hablamos con el mago. La variable *talkwizardlines* sirve para controlar qué frase deberá decir el mago dependiendo de si ya se ha hablado con él previamente o no; digamos que es una variable que si vale 1, por ejemplo, el mago sólo dirá unas determinadas frases disponibles, si vale 2 las frases anteriores ya no se volverán a decir nunca más y se pasa "a otro nivel" de frases, y así. Seguramente se podría haber hecho de una manera más óptima, te lo aseguro. Si se cumplen todas esas condiciones, aparecerá una frase concreta que dice el mago, se espera un cierto retardo para que haya tiempo de leer la frase y, una vez acabado el tiempo,la frase desaparece. El retardo viene dado por la inclusión del parámetro numérico en la orden Frame; recuerda que este parámetro fijaba cuántos fotogramas podían pasar sin que se mostrara para el proceso en cuestión ningún cambio en pantalla. En este caso, poniendo un valor como 5000, lo que estamos haciendo es esperar un tiempo igual a 50 frames para que el código del proceso pueda continuar ejecutándose (durante esos 50 frames, evidentemente,los otros procesos que pudieran existir habrán ejecutado su código de forma completamente normal). Finalmente, el significado de la variable *talking* la veremos más adelante (fíjate que se pone a true antes de mostrar la frase y cambia a false cuando se ha dejado de mostrar).

Todos los ifs son más o menos iguales,excepto el cuarto If, que es cuando Bob pregunta al protagonista si quiere o no la llave, y el protagonista ha de decidir si responde "Sí" o "No". En ese If aparece la llamada a un proceso que no hemos visto todavía, "talk_cursor()", y se introduce dentro de un bucle hasta que demos una respuesta.

Hagamos, antes de acabar, unas pequeñas modificaciones necesarias en diferentes partes del programa. Primero, al final de todo del Case 1, escribe las siguientes líneas:

```
MyChar_has_KeyOne=false;
near_bob=false;
near_wizard=false;
talkboblines=1;
talkwizardlines=1;
asked_bob_for_key=false;
```

Esto es porque cuando en algún momento se vuelva al menú inicial – o al poner en marcha el juego-, los valores de estas variables se reseteen y valgan su valor inicial por defecto otra vez.

Y otra cosa que tendremos que cambiar es añadir al principio del programa principal, donde se cargan las fuentes, la línea que carga nuestra nueva fuente para los diálogos.

```
talk_font=load_fnt("talk_fon.fnt");
```

Y otra cosa que hay que hacer, es evidentemente, llamar al proceso "talklines()" en algún momento para que se pueda ejecutar. Coincidirás conmigo que el lugar más natural es al inicio del Case 3. Justo después de haber creado el proceso "MyCharacter_in_House()", "Wizard()" y Bob()", seguidamente añade:

```
talklines();
```

Finalmente, crearemos el código para el proceso “talk_cursor()” que vimos que se creaba cuando Bob nos preguntaba si queríamos la llave y nuestro protagonista se veía obligado a responder. Fíjate que básicamente es el mismo proceso que el del cursor de la pantalla inicial; sólo cambia el gráfico y las coordenadas. En este caso, lo que apuntará el cursor será a las dos posibles respuestas que podamos dar: “Sí” o “No”.

```
PROCESS talk_cursor()
PRIVATE
    Int c;
END
BEGIN
    file=file1;
    graph=14;
    x=70;
    y=170;
    c=3;
    yes=false;
    no=false;

WHILE (NOT key(_space) AND NOT key(_control) AND NOT key(_enter) AND no==false and
yes==false)
        IF (key(_up))  y=170; END
        IF (key(_down)) y=190; END
        x=x+c;
        IF (x>=80) c=-3; END
        IF (x<=70) c=3;  END
        FRAME;
        IF (key(_esc))   no=true;  y=190;   END
    END

    SWITCH (y)
        CASE 170:  yes=true;   END
        CASE 190:  no=true;   END
    END
END
```

Y por último, explico el significado de la variable *talking*. Es buena idea hacer que nuestro personaje no se pueda mover mientras está conversando con los personajes. Así que el valor que contiene esta variable denota si en ese momento el personaje está hablando o no, y por tanto, podríamos utilizarla para inmovilizar al personaje en consecuencia. Lo único que tenemos que hacer para eso es modificar el proceso “MyCharacter_in_House()” forzando a que los movimientos los haga (a más a más de las restricciones anteriores) siempre que *talking* valga false. Es decir, tienes que modificar lo siguiente:

```
Process MyCharacter_in_House(x,y)

…

if(key(_right) and talking==false) despega_x=2; end
if(key(_left) and talking==false) despega_x=-2; end
if(key(_up) and talking==false) despega_y=-2; end
if(key(_down) and talking==false) despega_y=2; end

if (key(_right) and talking==false and obstacle==false)
…
end

if (key(_left) and talking==false and obstacle==false)
…
end

if (key(_up) and talking==false and obstacle==false)
…
end

if (key(_down) and talking==false and obstacle==false)
```

```
…
end

…
End
```

Ahora, por fin, podrás tener la conversación soñada con los dos personajes. Esta conversación es sólo de ejemplo: modifícala como creas oportuno.

## *Entrar y salir de la segunda casa

Lo único que vamos hacer ahora es añadir una nueva casa en el mapa verde, y hacer, en el Case 4, que podamos entrar en ella, siempre y cuando tengamos la llave que nos ha dado el mago.

Primero de todo, los gráficos:

-Dibuja otra casa en el mapa verde. Yo dibujaré esta segunda casa en la esquina superior izquierda del mapa verde, en el extremo.Vuelve a guardar el mapa con el mismo código, el 300.

-Modifica también el mapa de durezas de nuestro mundo. Ten encuenta una cosa muy importante: el color de la segunda casa en el mapa de durezas puede seguir siendo el mismo rojo (porque ambas casas son obstáculos igual), pero ahora el color que indica el umbral de la puerta para esta segunda casa ha de ser diferente del amarillo –el color que usamos en la primera casa-, porque si fuera igual, tal como hemos escrito el código, el personaje entendería que entra en la primera casa, y no en la segunda. Cuidado con esto. Así pues, elige otro color que no sea el amarillo –yo elijo el naranja- para indicar en el mapa de durezas que la puerta de entrada a la segunda casa es diferente. El código de ese color lo guardaremos cuando utilicemos el **map_get_pixel()** en la variable globar *colorUmbral2*, así que también tendrás que crear esta nueva variable.Vuélvelo a guardar con el mismo código de antes, el 102.

-Para el interior de la casa puedes utilizar el mismo dibujo que el de la primera casa, o bien otro dibujo. Nosotros no queremos trabajar mucho y usaremos el mismo dibujo,así que no tocamos nada y para la segunda casa seguiremos usando el dibujo número 401. Si tú quieres hacer otro dibujo, tendrás que usar otro código y modificar el programa.

-Crea un nuevo mapa de durezas para la segunda casa. Si usamos el mismo gráfico 401 para visualizar el interior de la casa, el mapa de durezas nuevo para esta segunda casa tendrá que tener la misma disposición espacial para los obstáculos, pintados en rojo.Lo que ha de cambiar respecto el mapa de durezas de la primera casa es el color con el que se denotará el umbral de la puerta de salida hacia el mundo, por el mismo motivo de antes. Si mantuviéramos el amarillo, el personaje saldría de la segunda casa pensando que sale de la primera y aparecería en el mundo verde al lado de ésta. Por tanto,hagamos que el color del umbral sea ahora otro. Mejor que sea el mismo que hemos utilizado en el mapa de durezas del mundo para indicar el umbral de entrada; o sea, naranja, y así podremos continuar utilizando la variable *color5*.Guarda este nuevo mapa de durezas con el código 103.

Añade la variable global entera *house2*.

Lo primero será modificar el Case 2 para que podamos entrar desde el mundo verde a la segunda casa. Justo antes del Loop del Case 2, escribe esto:

```
//Recoge el color de un punto del umbral de la puerta en el mapa de durezas
colorUmbral2=map_get_pixel(file1,102,125,55);
```

y justo antes de la línea *if(key(_esc))* escribe esto:

```
IF    (MyChar_has_KeyOne==true    and    map_get_pixel(file1,102,    idChar.x+despega_x,
idChar.y+despega_y)==colorUmbral2)
          house2=true;
ELSE
          house2=false;
```

```
END
```

Fíjate en la condición que se pone: el personaje ha de tener la llave para poder entrar.Y fíjate también lo que te he dicho antes: si hubiéramos dejado *colorUmbral1* como color del umbral de la casa, inmediatamente antes de este if que acabamos de poner está el if que comprueba si el personaje ha chocado o no con el umbral de la primera casa:si hubiéramos puesto el mismo color de umbral, el personaje no podría distinguir en qué casa entra.

Y en el mismo Case 2, justo después del bloque *if(house1==true)/end* –justo antes del último FRAME- escribe esto:

```
if(house2==true)
          house2=false;
          level=4;
          MyChar_Position_X=160;
          MyChar_Position_Y=60;
          break;
end
```

Evidentemente, ahora tendremos que modificar el Case 4 para que podamos entrar y salir de la segunda casa.El Case 4 quedará así,pues:

```
Case 4:
          put_screen(file1,401);
      idCharHouse=MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);

          Loop

IF(map_get_pixel(file1,103,idCharHouse.x+despega_x,idCharHouse.y+despega_y)==colorObstacu
lo)
                              obstacle=true;
                    ELSE
                              obstacle=false;
                    END

IF(map_get_pixel(file1,103,idCharHouse.x+despega_x,idCharHouse.y+despega_y)==colorUmbral2
)
                              house2=true;
                    ELSE
                              house2=false;
                    END

                    if(house2==true)
                              house2=false;
                              level=2;
                              MyChar_Position_X=300;
                              MyChar_Position_Y=300;
                              break;
                    end

                    if(key(_esc))
                              while(key(_esc)) frame; end
                              level=1;
                              break;
                    end

                    frame;
          End //Loop

fade_off();
delete_text(0);
clear_screen();
let_me_alone();
fade_on();
End
```

Un comentario final: en este tutorial ya no volveremos a retocar más el mapa de durezas, así que los cinco colores que tiene son los que tendrá. Aprovecho ahora, pues, para comentarte que sería una buena idea recopilar las cinco líneas esparcidas por todo el código fuente donde definíamos el color de cada una de estas partes (las líneas del tipo *colorX=map_get_pixel(...)*) y escribirlas juntas una detrás de otra, al inicio del programa principal, antes del bloque Loop/End. De esta manera, mantendrás una coherencia en el código porque sabrás que todas las asignaciones de colores para las distintas zonas del mapa de durezas se realizan al principio del juego, en el mismo sitio del código; y justo después de haber cargado el FPG es el lugar ideal. El código mejorará en claridad y comodidad. Y de hecho, incluso podrías hacer otra cosa más, que es crear un nuevo gráfico que sirviera simplemente como tabla de muestras de colores, y que a la hora de asignar los valores a la variables *colorX* se cogieran de allí en vez del mapa de durezas. Con esto ganarías facilidad a la hora de cambiar colores y/o posiciones de durezas dentro del mapa.

## *La pantalla del inventario

En este apartado vamos a darle a nuestro personaje una espada y hacer que la pueda usar cuando pulsemos ENTER. La idea (no tiene por qué ser la mejor) es que en cualquier momento del juego, si el jugador pulsa la tecla "I", ha de aparecer en la parte superior de la pantalla una zona en forma de barra que representará el inventario del protagonista. Este inventario sólo tendrá un hueco para incluir –o no- la espada; ya verás que es fácil hacer que pueda tener más huecos para incluir más elementos, pero lo tendrás que hacer por tu cuenta. La barra aparecerá por encima del gráfico del juego que estuviera visualizándose en ese momento (o sea, el personaje en el mundo verde o dentro de alguna casa), sin borrarlo pero dejándolo congelado, hasta que el jugador vuelva a pulsar "I", momento en el cual la barra superior desaparecerá, dejando otra vez toda la pantalla para nuestro personaje y dotándolo otra vez de movimiento y vida.

Vamos a hacer que nuestro personaje desde el principio del juego tiene la espada ya en su inventario. Y vamos a hacer que para que la pueda usar, el jugador tenga que forzar la aparición de la barra del inventario y seleccionar la espada de allí. Haciendo esto, el gráfico de nuestro personaje cambiará en el mundo y se le verá manejando la espada, y el inventario en ese momento tendrá que aparecer vacío. A partir de entonces, si se vuelve a apretar "I", se podrá seguir jugando con la espada "desenfundada". Cuando el jugador desee, pulsará "I" otra vez y podrá devolver la espada al inventario haciendo que el protagonista "enfunde" la espada.

Primero, los gráficos:

-Haz un fondo blanco para la barra del inventario, de 320x40 píxeles. Guardala con el código 216.

-Crea una imagen de 20x20 píxeles y dibuja un cuadrado gris vacío en ella.Guárdala con el código 217.Este cuadrado gris representará el –único- hueco del que dispondrá nuestro inventario; es decir, representa la falta del único elemento que podremos incluir (la espada). De hecho, en el interior de este cuadrado podrá aparecer el ítem espada, o no, dependiendo de si en ese momento el protagonista la está utilizando. Si quisieras que en el inventario pudieran caber más elementos ("ítems"), tendrías que modificar el código que verás próximamente para añadir más imágenes de este cuadrado gris, denotando así que hay sitio para más objetos dentro de él.

-Crea una imagen idéntica a la anterior, pero con una espada dentro. Guárdala con el código 218. Éste será el gráfico que se mostrará en sustitución del anterior si la espada viene incluida dentro del inventario del personaje (es decir, si el hueco no está vacío).

-Crea una imagen idéntica a la anterior, pero con el cuadrado de color azul. Guárdala con el código 221. Éste será el gráfico que se muestre cuando apretemos la tecla ENTER en la pantalla del inventario, y denotará que hemos elegido el objeto del interior del hueco –la espada- para cogerla y usarla en el juego

-Crea cinco imágenes de nuestro protagonista mirando hacia el sur (20x20 también), empuñando la espada. En cada imagen la orientación de la espada ha de ser diferente, dando la sensación en conjunto de estar moviendo la espada de derecha a izquierda y viceversa. Guarda estas imágenes con los códigos de 700 a 704.

-Haz lo mismo que en el punto anterior pero con el personaje mirando al norte. Guarda estas imágenes con los códigos del 705 al 709. Y si quieres también, aunque yo no lo haré, puedes hacer lo mismo pero mirando hacia el este.

Lo siguiente que haremos, antes de continuar, es añadir a la lista una cuantas variables globales enteras más, que usaremos a lo largo de todo este apartado:

*to_item_select_screen*
*last_level*
*direction*
*kill_mode*
*sword_selected*

Y también añadiremos una variable global entera más, pero que deberá ser inicializada a true: *sword_in_inventory*; porque recuerda, y esto es muy importante, que si cuando creamos una variable entera, no la inicializamos (como las cinco anteriores), Bennu automáticamente la inicializará a 0, o lo que es lo mismo, a falso. Acuérdate de esto si no quieres tener sorpresas

Una vez creados los gráficos y declaradas (e inicializadas en su caso) las variables globales necesarias, empecemos con el código a escribir. Lo primero que haremos es poner en el Case 2, Case 3 y Case 4 que para ir al Case 5 ( y mostrar así la pantalla del inventario por encima de las otras) habrá que pulsar la tecla "I".

Modificaremos los Cases así. En el Case 2:

```
Case 2:
          if (to_item_select_screen==false)
                    idChar=MyCharacter(MyChar_Position_X,MyChar_Position_Y);
                    start_scroll(0,file1,300,0,0,0);
                    color=map_get_pixel(file1,102,1500,1500);
             colorUmbral1=map_get_pixel(file1,102,1525,1535);
             color5=map_get_pixel(file1,102,125,55);
          end
          to_item_select_screen=false;
          Loop
                    …
                    if(house2==true)
                              house2=false;

                              level=4;
                              MyChar_Position_X=160;
                              MyChar_Position_Y=60;
                              break;
                    end

                    if (key(_i))
                               level=5;
                               break;
               end
                    frame;
          End
          if (level<>5)
                     fade_off();
                     delete_text(0);
                     let_me_alone();
                     stop_scroll(0);
                    fade_on();
          end
          last_level=2;
End
```

En el Case 3, haremos los mismos cambios:

```
Case 3:
          if (to_item_select_screen==false)
                    put_screen(file1,401);
               idCharHouse=MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);
                    Wizard(250,105);
```

```
                    Bob(130,100);
                    talklines();
                    colorMago= map_get_pixel(file1,106,250,105);
                    colorBob= map_get_pixel(file1,106,130,100);
          end
          to_item_select_screen=false;
          Loop
                    …
                    if(house1==true)
                              house1=false;
                              level=2;
                              MyChar_Position_x=1516;
                              MyChar_Position_y=1549;
                              break;
                    end
            if (key(_i))
                              level=5;
                              break;
                    end
          if(key(_esc))
                              while(key(_esc)) frame; end
                              level=1;
                              break;
                    end
                    frame;
          End
          if (level<>5)
                     fade_off();
                     delete_text(0);
                     clear_screen();
                     let_me_alone();
                     fade_on();
          end
          last_level=3;
End
```

Y en el Case 4, igual:

```
Case 4:
          if (to_item_select_screen==false)
                    put_screen(file1,401);

          idCharHouse=MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);
          end
          to_item_select_screen=false;
          Loop
                    …
                    if(house2==true)
                              house2=false;

                              level=2;
                              MyChar_Position_X=170;
                              MyChar_Position_Y=100;
                              break;
                    end
                    if (key(_i))
                               level=5;
                               break;
                    end
                    if(key(_esc))
                              while(key(_esc)) frame; end
                              level=1;
                              break;
                    end
                    frame;
          End //Loop
          if (level<>5)
```

```
                        fade_off();
                        delete_text(0);
                        clear_screen();
                        let_me_alone();
                        fade_on();
            end
            last_level=4;
End
```

Antes de explicar los cambios que hemos introducido en los tres Case anteriores, que son muy sutiles e importantes, voy a escribir a continuación cómo ha de ser el código del Case 5, para que puedas ver de una sola vez todas las partes del código de los diferentes Cases que están relacionadas entre sí. Así que el Case 5 ha de ser así:

```
Case 5:
            delete_text(0);/*Si en la pantalla queda algún diálogo visible,que se borre*/
            to_item_select_screen=true;

            item_select_screen_background();
            signal(idChar,s_freeze);
            signal(idCharHouse, s_freeze);

            if (sword_in_inventory==true)
                        item_select_cursor();
            else
                        item_select_cursor_empty();
            end

            frame(2000);

            Loop
                        if (key(_i))
                                    level=last_level;
                                    break;
                         end
                        frame;
            End

            signal(type item_select_screen_background,s_kill);
            signal(type item_select_cursor,s_kill);
            signal(type item_select_cursor_empty,s_kill);

            signal(idChar,s_wakeup);
            signal(idCharHouse,s_wakeup);

            frame(2000);

End
```

Comencemos. ¿Qué hemos modificado en los Case 2,3 y 4? Lo que es más evidente de entender es que se ha incluido la posibilidad dentro del Loop –para que se vaya comprobando constantemente- de que se pulse la tecla "I" para salir del bucle e ir al Case 5. Ahora miremos el código del Case 5. Fíjate que para salir de este nivel también se tendrá que pulsar la tecla "I". Cuando ocurra esto, se le asigna a la variable *level* –la que se usa en la condición del switch para saber a qué nivel se va- el valor de otra variable, *last_level*.¿De dónde ha aparecido esta nueva variable? Fíjate que otra de las modificaciones de los Case 2,3 y 4 es que justo antes de salir de ellos guardamos en esta variable el nivel donde estamos (puede ser el nivel 2,3 o 4).De esta manera, cuando queramos salir del Case 5, *last_level* valdrá el nivel donde estábamos antes de entrar en la pantalla del inventario, y por tanto, al salir del inventario sabremos a qué nivel tenemos que ir.

También salta a la vista los varios retardos que hemos introducido en el Case 5, para que el dedo tenga tiempo de soltar las teclas ("I",etc)de forma adecuada.

En los Case 2,3 y 4 hay dos modificaciones más. Al principio, la inclusión de la creación de fondos y la de los distintos procesos dentro del bloque *if (to_item_select_screen==false);* y al final, la inclusión de las últimas líneas de los Case dentro del bloque *if(level<>5)*. Estas modificaciones requieren de un estudio profundo para saber su razón de ser. Primero miremos para qué sirve la variable *to_item_select_screen*, inicializada en principio a false. Vemos

que esta variable toma el valor “true” nada más entrar en el Case 5, siempre.Y que una vez salido de éste, continúa manteniendo su valor. Por tanto, esta variable no es más que una marca, una señal que indica que si vale “true” se ha pasado por el Case 5 al menos una vez. Si valiera “false”, en principio podemos pensar que no se ha entrado en la pantalla del inventario todavía. ¿Y para qué sirve saber si se ha entrado o no en el inventario?

Recordemos lo que queríamos hacer: poner la pantalla del inventario en forma de barra por encima de la pantalla que estuviera visible en ese momento, en la parte superior de la ventana. Fíjate qué es lo que ponemos dentro del if: básicamente la creación de los procesos (personaje, personaje_en_casa, mago,bob…).Si no se ha entrado en el inventario, se crean normalmente, pero si hemos entrado, no se crearán.¿Por qué? ¡Porque ya están creados¡ Piensa un poco. Cuando estemos en la ventana del inventario, los procesos que estuvieran en ese momento visible continuarán estándolo,tal como hemos dicho. Así pues, cuando salgamos del Case 5 porque hemos “cerrado” la pantalla del inventario y queramos seguir jugando allí donde estábamos, lo que hará nuestro programa es volver al Case adecuado (2,3 o 4) y volver a ejecutar su interior. Si no pusiéramos este if, por tanto, se volverían a crear otra vez los procesos que no han sido destruidos, por lo que tendríamos dos personajes, o dos personajes_en_casa, dos magos,etc… Y así, cada vez que entráramos en el inventario, crearíamos nuevos procesos. Prueba de quitar el if y verás lo que te comento más claramente.

Una vez evitado el escollo de crear nuevos procesos que no deben crearse, fíjate que hemos vuelto a poner en los tres Cases el valor de *to_item_select_screen* a false, para volver a empezar como si no hubiera pasado nada. Si no, la variable *to_item_select_screen* valdría siempre true y nos podría dar problemas.

¿Y para qué sirve el otro if, el del final? Fíjate qué es lo que hemos incluido en este if: el fade_off, el borrado de cualquier texto que hubiera, el borrado de la imagen de fondo –ya sea estática mediante **clear_screen()** o escrolleable mediante **stop_scroll()**- el fade on y, muy importante, la función **let_me_alone()**. Esta función es de vital importancia para entender lo que estamos haciendo, y hasta ahora no le habíamos prestado ninguna atención y ni tan sólo sabíamos por qué la poníamos. Ya es hora de remediarlo.

Esta función está en los tres Cases:(2,3 y 4). La tarea de este comando es matar todos los procesos excepto el proceso que lo llama, que en nuestro caso es el programa principal. Piensa un poco: si estuviéramos ejecutando el código del Case 2, ¿qué procesos habría activos en ese momento? Aparte del proceso principal, es evidente que los procesos que hubiéramos creado en el propio Case 2, es decir, “MyCharacter()”. Por lo tanto, si salimos del Case2, con **let_me_alone()** mataríamos el proceso MyCharacter. Haciendo un análisis similar podríamos ver que saliendo del Case 3 mataríamos los procesos MyCharacter_in_House()”, ”Wizard()”, ”Bob()” y “Talklines()”; y saliendo del Case 4 mataríamos el proceso “MyCharacter_in_House()”.¿Y por qué hacemos esto? Está claro: si entramos en un nivel, no nos interesa para nada tener los procesos de otro nivel funcionando: si estamos en el mundo verde no queremos para nada el mago, ¿verdad? Y además, haciendo esto nos facilitamos las cosas porque como habremos destruido al salir los procesos, al regresar a ese nivel los volvemos a crear y ya está. De hecho, podrías probar el quitar esta línea y verás que cada vez que volvamos a un nivel donde ya hubiéramos estado antes nos aparecería un nuevo personaje, o personaje_en_casa, o mago,etc…porque no habríamos matado el anterior.

Pues bien, la explicación del bloque *if(level<>5)* está clara. Queremos que ocurra el proceso que he descrito en el párrafo anterior siempre que nos movamos entre los Case 2,3 y 4 (que entremos a o que salgamos de la casa 1 o 2 al mapa verde y viceversa), pero no cuando desde cualquier de estos niveles vayamos al nivel 5. ¿Por qué? Porque como hemos dicho que pondremos una barra superior en la ventana pero que la pantalla seguirá viéndose tal como estaba, no podremos matar los procesos “MyCharacter()” y compañía, porque si no lo que se vería sería la barra, sí, pero sobre un fondo negro. Por tanto, siempre que vayamos a la pantalla del inventario, no mataremos los procesos existentes ya que queremos que continúen en la pantalla.

No obstante, lo que sí que haremos será que estos procesos, aunque visibles, no se puedan mover ni puedan responder a ninguna acción del jugador, mientras la ventana del inventario esté activada (quedaría muy feo): haremos un “pause” del juego. ¿Y ésto cómo lo hacemos? Podríamos utilizar el mismo sistema que hicimos para las conversaciones entre el protagonista y el mago y bob: hacer que mientras el valor de una variable (en aquel caso, *talking*) sea uno determinado, las condiciones de los ifs del movimiento hagan que el personaje no se pueda mover. No obstante, ahora usaremos otra técnica un poco más sofisticada. Lo que haremos será que cuando entremos en el Case 5 (la pantalla del inventario activada), congelemos los procesos susceptibles de moverse, es decir, “MyCharacter()” y “MyCharacter_in_House()”. Acuérdate que congelar significa que el gráfico de ese proceso permanecerá en pantalla pero que su código no se ejecutará y por tanto permanecerá impasible, hasta que “se despierte”. Y acuérdate que congelar y despertar lo podíamos hacer con la orden **signal()**. Acuérdate de que como primer parámetro de esta función podíamos poner o bien el identificador concreto de un proceso, o bien indicar los procesos que fueran de un tipo determinado.Pues bien, si te fijas, eso es lo que he hecho en el Case 5: congelar los dos procesos, y luego, antes de salir

del Case 5, despertarlos otra vez.

Un detalle, el proceso “MyCharacter()” solamente estará activo en el Case2 y “MyCharacter_in_House()” lo estará en el Case 3 y el 4. Dentro del Case 5 podría haber escrito un if que comprobara el valor de *last_level* y si éste era igual a 2, hacer que se congelara sólo “MyCharacter()”, y si éste era igual a 3 o 4, hacer que se congelara sólo “MyCharacter_in_House()”. No obstante, Bennu no da ningún error cuando se le dice que se envíe una señal a un proceso inexistente…así que nos despreocupamos de este hecho.

Falta todavía por repasar qué es lo que hace en definitiva el Case 5, aparte de congelar y descongelar los procesos existentes en el nivel de donde se procede. Lo primero que ejecuta es un proceso creado por nosotros también llamado “item_select_screen_background()”.Su código es el siguiente:

```
Process Item_Select_Screen_Background()
begin
          graph=216;
          X=160; //Fondo centrado
          z=-5;
          loop
                    frame;
          end
end
```

Puedes comprobar fácilmente que lo único que hace este proceso –de hecho, la razón de su existencia- es poner un gráfico, que será una barra visible en la parte superior de la ventana, la cual representará la pantalla del inventario. Como queremos que esta barra se vea encima del gráfico de fondo visible hasta ahora (el mapa verde, alguna casa), hemos de especificarle un valor de la variable Z adecuado.Fíjate que este proceso es infinito (el bucle no tiene ninguna condición de finalización), por lo que en principio esta barra, una vez que se vea, se verá hasta la finalización del juego. Para que esto no ocurra, fíjate que en Case 5, después de que el jugador haya pulsado “I” para hacer desaparecer el inventario de su vista, se le envía una señal S_KILL a este proceso para matarlo.

Seguidamente nos encontramos con un bloque *if/else* que comprueba si el valor de la variable *sword_in_inventory* es true o no. Si lo es, se ejecutará otro proceso creado por nosotros, “item_select_cursor()”, y si no lo es, se ejecutará el “item_select_cursor_empty()”. Acuérdate de que la variable *sword_in_inventory* la inicializamos a true por defecto. Esta variable indicará si la espada está (true) o no (false) en el inventario. Es decir, que si vale true, deberá aparecer en la barra del inventario un gráfico indicando que la espada está disponible para poder cogerla, y si vale false, la barra del inventario aparecerá vacía, y –esto lo veremos en seguida- nuestro protagonista deberá aparecer empuñando la espada. Como puedes ver, hemos dicho a la computadora que tienes ya la espada de entrada , pero puedes poner esto a falso, y entonces hacer algo en el juego para que sea verdadero (en otras palabras, puedes conseguir la espada en una tienda o similar).

Así pues, si la espada está en el inventorio, se ejecuta “item_select_cursor()”. Si no, “item_select_cursor_empty()”.El primer proceso es simplemente una manera para mostrar el gráfico de la espada dentro de la barra del inventario (evidentemente, si la espada no estuviera, no deberíamos mostrar su correspondiente gráfico en el inventario, por lo que no deberíamos ejecutar este proceso). Una cosa muy importante que debes tener presente es que como en este tutorial sólo tenemos un item en el inventario, el código de este proceso es bastante trivial, pero si quieres tener más elementos, deberás de jugar con las coordenadas de cada item en la barra, y utilizar los cursores para poder seleccionar el item adecuado, comprobando si está o no está en ese momento…Por lo tanto, el código de este proceso puede complicarse un poco. El código de este proceso es el siguiente:

```
Process Item_select_cursor()
Begin
          graph=218;
          x=10;
          y=10;
          z=-15;

          loop
                    if(key(_enter) and graph=218)
                              graph=221;
                              sword_selected=true;
                              frame(1000);
                    end
```

```
                    if(key(_enter) and graph=221)
                              graph=218;
                              sword_selected=false;
                              frame(1000);
                    end
                    frame;
          end
End
```

Puedes comprobar que si se apreta la tecla ENTER, el cuadrado que rodea al gráfico de la espada (el cursor de selección, que en este caso siempre seleccionará al único ítem existente) cambia de apariencia. Esta tecla servirá pues para seleccionar un ítem y poder utilizarlo en el juego.En seguida lo veremos.

El otro proceso lo único que hace es mostrar el mismo cuadrado gris que antes incluía la imagen de la espada, pero ahora gris. Y no podemos interaccionar nada con él: la barra del inventario se muestra desierta e inútil. Aquí está su código.

```
Process Item_select_cursor_empty()
Begin
          graph=217;
          x=10;
          y=10;
          z=-15;
          loop
                    frame;
          end
end
```

Si ejecutas en estos momentos el juego, verás que aparece la barra de inventario y podemos apretar ENTER, pero si salimos del inventario no ha pasado nada de nada. Tenemos que hacer que si se selecciona la espada del inventario, al volver al juego nuestro protagonista la empuñe y la pueda mover de un lado a otro, por ejemplo.Manos a la obra. Vamos a hacer que cuando seleccionemos con la tecla ENTER en el inventario la espada, y salgamos de la pantalla del inventario, dentro del mundo o de las casas el protagonista pueda mover la espada cuando se pulse por ejemplo la tecla "e". Habrá que modificar dos procesos: "MyCharacter()" y "MyCharacter_in_House()", de esta manera:

```
Process MyCharacter (x,y)
BEGIN
          …
          Loop
                    …
                    if(key(_right)) despega_x=2; direction=2;end
                    if(key(_left)) despega_x=-2; direction=4;end
                    if(key(_up)) despega_y=-2; direction=1;end
                    if(key(_down)) despega_y=2; direction=3;end

                    …

                    if   (sword_selected==true   and   key(_e)   and   direction==1   and
kill_mode==false)                   graph=705;
                              kill_mode=true;
                      repeat
                               graph=graph+1;
                               frame;
                      until (graph==709)
                    graph=6;
                    else
                          kill_mode=false;
                    end

                    if   (sword_selected==true   and   key(_e)   and   direction==3   and
kill_mode==false)
                              graph=700;
                              kill_mode=true;
                      repeat
```

```
                        graph=graph+1;
                            frame;
                    until (graph==704)
                            graph=1;
                else
                        kill_mode=false;
                end


                frame;
      End //Loop
End
```

Para el proceso "MyCharacter_in_House()" se deberán hacer exactamente las mismas modificaciones.

No hay mucho que comentar: siempre que se pulse la tecla "e" y se haya escogido la espada del inventario, aparecerá la animación corta de nuestro personaje empuñando la espada, orientado hacia una dirección u otra.Para saber la orientación de la animación (si son los gráficos mirando al norte, al sur,etc) se utiliza la variable *direction*. En este código no hemos escrito las animaciones para las orientaciones este y oesta: tendrías que escribir dos ifs similares a los mostrados, cambiando los códigos de los gráficos adecuados en el FPG.

Finalmente, un detalle que no podemos escapar es resetear el valor de algunas variables para que si el jugador desea volver a empezar el juego habiéndolo empezado ya (es decir, volver al Case 1 desde cualquier otro Case), que el jugador no se encuentre empezando el juego con la espada empuñada, por ejemplo.Así que añade las siguientes líneas al final del Case 1:

```
to_item_select_screen=false;
last_level=0;
sword_in_inventory=true;
kill_mode=false;
sword_selected=false;
```

Un último detalle para que trabajes…Fíjate que si ejecutas el juego y seleccionas la espada del inventario, la podrás usar en el juego, pero si vuelves al inventario, el dibujo de la espada todavía continúa allí como si no la hubiéramos cogido. Una solución sería poner esto

```
If(sword_selected==true)
          Sword_in_inventory=false;
End
```

en el Case 5, justo antes de las señales S_KILL. No obstante, queda pendiente un hecho:si en este momento nuestro personaje empuña la espada,¿cómo podemos devolver ésta al inventario otra vez? Queda como ejercicio.

Por cierto, la variable *kill_mode* se usará próximamente, cuando añadamos enemigos.

## *Guardar partidas en el disco duro y cargarlas posteriormente

Lo primero que tienes que decidir es cuáles serán las variables cuyos valores necesitas guardar. La idea es que apretando CTRL+S o algo parecido se guarden dichos valores para posteriormente cargarlos apretando CTRL+L o similar y proseguir el juego con los valores almacenados de estas variables. Aquí os pongo la lista de las variables que vamos a guardar:

level
MyChar_Position_X
MyChar_Position_Y
MyChar_has_KeyOne
talkboblines
talkwizardlines

asked_bob_for_key
sword_selected
sword_in_inventory

Guardaremos estos valores en una tabla. En la lista hay 9 variables, pero a lo mejor querrás salvar más variables más adelante, así que crearemos una tabla llamada *savedata[20]* con 21 elementos, incluyendo el 0.Así que añade una variable global entera más a la lista, que será dicha tabla:

```
int savedata[20];
```

Y ahora tendremos que modificar el Case 1 de la siguiente manera:

```
Case 1:
            …
            Loop
                        if(selection==1) level=2; break; end;
                        if(selection==2) fade_off();exit("Gracias por jugar");end

               if(key(_control) and key(_s))
                      savedata[0]=level;
                      savedata[1]=MyChar_Position_X;
                      savedata[2]=MyChar_Position_Y;
                      savedata[3]=MyChar_has_KeyOne;
                      savedata[4]=talkboblines;
                      savedata[5]=talkwizardlines;
                      savedata[6]=asked_bob_for_key;
                      savedata[7]=sword_selected;
                      savedata[8]=sword_in_inventory;

                      save("player.dat",savedata);
                       end
                       frame;
            end
…
End
```

Lo único nuevo aquí es el uso de la función ya conocida **s*ave()***. Este comando, recuerda, sirve para guardar el contenido de una variable (o estructura, o tabla) en un determinado archivo, el cual se puede crear en ese momento, o utilizarse uno ya existente, borrándose lo que contuviera antes –así que ojo-. El primer parámetro de ***save()*** es precisamente el nombre completo (ruta más nombre) del archivo donde vamos a grabar los datos; si no se pone ruta el archivo se creará (o buscará si ya existe) en la misma carpeta donde esté el DCB.La extensión de este archivo puede ser cualquiera.Y el segundo parámetro es el nombre de aquella variable, estructura o tabla que se quiere guardar.

Para ver si funciona, vamos ha hacer que nuestro juego pueda cargar los valores de las variables previamente guardadas. Para ello, tendremos que utilizar la función **l*oad()***. Esta función tiene los mismos dos parámetros: la ruta completa de un archivo y el nombre de una variable/tabla/estructura, pero en este caso lo que hace es "leer" el archivo especificado y rellenar con los datos leídos la variable/tabla/estructura que se le haya dicho. Comprenderás que es importante,que carguemos con **l*oad()*** las mismas variables/tablas/estructuras que fueron usadas a la hora de grabar, porque si no la información leida no tendrá ningún sentido. En nuestro juego haremos que se carguen los valores pulsando CTRL+L, así que, en el Case 1, justo después del bloque *if(key(_control) and key(_s))/end* –antes del Frame;- añadiremos lo siguiente:

```
if (key(_control) and key(_l))
            load("player.dat", savedata);

            level=savedata[0];
            MyChar_Position_X=savedata[1];
            MyChar_Position_Y=savedata[2];
            MyChar_has_KeyOne=savedata[3];
            talkboblines=savedata[4];
            talkwizardlines=savedata[5];
            asked_bob_for_key=savedata[6];
            sword_selected=savedata[7];
```

```
            sword_in_inventory=savedata[8];
            break;
end
```

Fíjate que una vez cargados en memoria, en la tabla *savedata* los valores leidos del archivo "player.dat",lo que hay que hacer es asignarle el valor de cada elemento recogido a la variable que le pertoca para que pueda ser usada en el programa con el nuevo valor. Y ya está.

Ahora, lo que hay que hacer es escribir estos dos bloques de ifs seguidos, el *if(key(_control) and key(_s))*/end y el *if(key(_control) and key(_l))/end* en los lugares oportunos de los Case 2, 3 y 4 (en el Case 5, evidentemente, no hace falta). Coincidirás conmigo en los sitios idóneos: en el Case 2 hay que escribirlo justo antes del Frame; del Loop; en el Case 3 antes del bloque *if(key(_esc))/end* que está al final del Loop; y en el Case 4,igualmente, justo antes del *if(key(_esc))/end* que está al final del Loop.

Testea tu programa. Por ejemplo, consigue la llave del mago y salva pulsando CTRL+S y entonces reinicia el juego. Pulsa CTRL+L y deberías de tener la llave y poder entrar en la segunda casa.

Lo más normal es que quieras hacer en tus juegos del estilo del que estamos haciendo un inventario que pueda incluir más de un sólo ítem. En este caso, la idea general sería la misma que la que hemos comentado en los párrafos anteriores, y sólo tienes que tener en cuenta unos detalles adicionales.Por ejemplo, para gestionar los diferentes elementos del inventario, lo más práctico es crear (con la sentencia TYPE) un tipo de datos personalizado "item", en el cual básicamente se guardará el tipo de ítem que es, el número de este ítem que hay actualmente en el inventario,etc. Haciéndolo así, el inventario como tal simplemente podría ser una tabla de estos ítems incluida dentro de una estructura que contendría éste y otros datos relevantes posibles relativos a nuestro personaje. De esta manera, podríamos controlar por ejemplo, para ese personaje, si tal elemento o tal otro de esa tabla (es decir, tal o cual ítem) está presente o no, consultando para ello el valor de dicho elemento: si vale 0 no hay ítem, y si vale otra cosa, entonces se podría saber qué ítem es consultando su campo identificador correspondiente. Es decir, algo parecido a esto:

```
type item
       int id=0; // identificador del objeto
       byte numero=0; // numero de unidades que tienes del objeto
end
```

Y...

```
struct player
    ...
    item inventario[X][Y];
end
```

donde X sería el número de columnas que tendrá el inventario e Y el número de filas del inventario

Esto por lo que respecta a la gestión lógica del inventario. Otro tema es cómo se visualizaría ese inventario en la pantalla. ¿Cómo hacer que cada objeto que coja el personaje, aparezca en el inventario cada en distintas coordenadas como si estuviesen ordenados? Habría que obtener las coordenadas del primer cuadro del inventario, y a partir de ahí sumar a cada objeto la diferencia de X. Una vez se haya llegado al extremo derecho de la ventana del inventario, se tendría que resetear la posición horizontal como al principio y sumar la diferencia de Y. Me explico: imagina por ejemplo que el centro del cuado de la espada es X=50 e Y=150, y que cada cuadro siguiente está 50 píxeles hacia la derecha y los de abajo están 30 píxeles por debajo. Si lo hacemos así, el primer ítem del inventario iría en (50,150), pero el segundo en (100,150), el tercero en(150,150), etc.Para colocar el objeto en el inventario, tendrías que saber cuántos hay (para conocer qué posicion le toca) y utilizar alguna fórmula parecida a ésta:

```
x=50+(50*idobjeto);
y=150;
```

donde decimos que la posición horizontal del ítem viene dado por su identificador (1,2,3...), colocándose así de forma ordenada respetando los tamaños y posiciones de los cuadros. Cuando se colocan tantos objetos que el siguiente debe ir debajo, y a hemos dicho que entonces la X tendría que resetearse y aumentar la Y. ¿Pero cómo se sabe cuándo ocurre esto? En las siguiente dos líneas solucionamos este problema y lo tenemos todo hecho de un golpe:

```
x=50+(50*(idobjeto%4));
```

```
y=150+(30*(idobjeto/4)));
```

Lo que hacemos aquí es utilizar la operación de módulo y aprovechar que la división de dos enteros devuelve un entero truncado, para dibujar un inventario que conste de cuatro cuadros por fila. Compruébalo "a mano" observando qué valores coge X e Y cuando idobjeto vale 1,2,3,4,5,etc.

## *Añadiendo enemigos, medida de vida y Game Over

Éste es el último apartado. Lo que haremos aquí es crear un proceso que muestre cuánta vida tiene el protagonista y que cuando se leacabe la vida haga volver automáticamente al jugador a la pantalla inicial (Case 1).

Primero los gráficos:

-Dibuja una flecha roja apuntando hacia arriba. Se pretende que sea un enemigo; lo puedes hacer mejor más adelante. Dibuja cuatro imágenes más de esta flecha roja, que serán las que se mostrarán cuando el enemigo muera. Guarda el enemigo "sano" con el código 600, y los otros cuatro con los códigos del 601 al 604.

-Dibuja 15 imágenes de 80x30 del medidor de vida. El primero debería representar la vida completa y el último debería de estar totalmente vacío. Puedes usar una barra o algo parecido a corazones, lo que tú quieras. Guárdalos entro del FPG con los códigos 200 al 214 (repito: el 200 sería el gráfico de máxima vida y el 214 el de nada de vida).

Ahora, a programar. Vamos a hacer un enemigo muy simple, el cual se mantiene siempre mirando hacia nuestro protagonista (como el mago y Bob) y además, que caminará detrás del personaje si éste se le acerca demasiado. Añade el siguiente proceso al final del programa.

```
Process Enemy_Pointer(x,y)
private
        id2;    //Esta variable contendrá el código identificador del protagonista
        idangle; //Ésta contendrá el ángulo entre el enemigo y el protagonista
        iddist; //Y ésta lo mismo que la anterior pero para la distancia
        hurt=3; //Vida del enemigo
        counter_kill=16;//Contador que retrasa la pérdida de vida del enemigo
        move_x;
        move_y;
end

begin
        z=1;
        ctype=c_scroll;//Estos enemigos se verán dentro del mundo verde
        graph=600;
        loop
                id2=get_id(TYPE MyCharacter);
/*La función get_dist() , como recordarás, lo que hace es devolver la distancia en píxeles entre el centro del gráfico del proceso identificado por "id2" –según la línea anterior, nuestro protagonista- respecto al centro del gráfico del proceso que realiza la llamada a get_dist(), o sea, "enemy_pointer". Resultado: almacena en "iddist" la distancia entre el protagonista y este enemigo.*/
                iddist=get_dist(id2);
                if (iddist=<81)
                        idangle=get_angle(id2);
                else
                        angle=0;
                end

/*Esta línea es la que hace que el enemigo esté siempre "mirando" al protagonista*/
                angle=idangle-90000;
```

```
/*Recuerda que las funciones get_distx y get_disty lo que hacen es devolver, en las mismas unidades que uses para la
distancia,el ancho y el alto respectivamente del rectángulo formado por la línea que recorre esa distancia (es decir: la
línea sería la diagonal de ese rectángulo y el ancho y alto del rectángulo serían los catetos de cualquiera de los dos
triángulos rectángulo que se formarían).Usando estas funciones puedes saber qué cantidades sumar a las coordenadas
X e Y de un objeto para desplazarlo en la dirección que desees.Repasa el apartado correspondiente del capítulo donde
se explicaron dichas funciones..*/

                        move_x=get_distx(idangle,iddist);
                        move_y=get_disty(idangle,iddist);

/*Los cuatro ifs de a continuación simplemente le dicen al enemigo que se mueva hacia nuestro personaje si éste se
hacerca demasiado al enemigo.*/

                        if ((move_x<-20 and move_x>-100) and (move_y>-100 and move_y<100))
                                x=x-1;
                        end
                        if ((move_x>20 and move_x<100) and (move_y>-100 and move_y<100))
                                x=x+1;
                end
                        if ((move_y<-20 and move_y>-100) and (move_x>-100 and move_x<100))
                           y=y-1;
                        end
                        if ((move_y>20 and move_y<100) and (move_x>-100 and move_x<100))
                           y=y+1;
                        end

/*Las siguientes líneas están para que el protagonista no pueda matar al enemigo tan rápidamente con la espada*/
                        counter_kill=counter_kill-1;
                        if (counter_kill==0)
                                counter_kill=16;
                        end

/*Si nuestro protagonista usa la espada, quitará alguna vida al enemigo. Fíjate que esto sólo lo podrá hacer si
"kill_mode" vale true.¿Te acuerdas de cuándo le asignamos ese valor?*/
                        if (collision (type MyCharacter) and kill_mode==true and counter_kill==1)
                                hurt=hurt-1;
                                x=x+get_distx(idangle,-10);
                                y=y+get_disty(idangle,-10);
                        end

                        if (hurt==0); //Si el enemigo muere, aparece la animación pertinente
                                repeat
                                        graph=graph+1;
                                        frame;
                                until (graph>604)
                                break;
                        end
                frame;
                end //Loop
end
```

Cuando hayas añadido este proceso tendrás que hacer otro para mostrar el medidor de vida, allí donde los valores especificados de sus variables locales ***X*** e ***Y*** digan (yo lo pondré en la parte inferior de la pantalla).Añade el siguiente proceso al final del programa:

```
Process Health_Meter(x,y)
Begin
           loop
                        If (health==14) graph=200;end
                        If (health==13) graph=201;end
```

```
                        If (health==12) graph=202;end
                        If (health==11) graph=203;end
                        If (health==10) graph=204;end
                        If (health==9) graph=205;end
                        If (health==8) graph=206;end
                        If (health==7) graph=207;end
                        If (health==6) graph=208;end
                        If (health==5) graph=209;end
                        If (health==4) graph=210;end
                        If (health==3) graph=211;end
                        If (health==2) graph=212;end
                        If (health==1) graph=213;end
                        If (health<=0) graph=214;end
                        frame;
            end
end
```

Como puedes comprobar, utilizo una variable llamada *health* que controla este proceso, por lo que habrá que añadirla también a la lista de variables globales enteras del programa.

Ahora nos falta llamar a este proceso en los Case 2,3 y 4.

Así pues, en el Case 2 escribiremos, justo después de iniciar el scroll con **start_scroll()** (dentro del bloque *if(to_item_select_screen==false)/end*):

```
health_meter(40,170);
```

En el Case 3 escribiremos la misma línea anterior, justo después de haber creado el proceso "talklines()". Y en el Case 4 la escribiremos justo después de la llamada al proceso "MyCharacter_in_House()".

Después de esto, lo que falta es modificar los dos procesos de nuestro protagonista , "MyCharacter()" y "MyCharacter_in_House()", para decirle que tendrá que perder vida si choca contra un enemigo.

Justo antes del Frame; del Loop, en ambos procesos, escribe esto:

```
atraso=atraso+1;
if (atraso==50)
            atraso=0;
end
if (collision(type Enemy_Pointer) and atraso==0)
            health=health-1;
end
```

Una nueva variable ha sido usada, atraso, así que tendrás que declararla en la lista de variables globales enteras. Fíjate que lo que hemos hecho ha sido simplemente hacer que cada vez que ocurra una colisión con un proceso de tipo enemigo,nuestro protagonista disminuya en 1 su energía. ¿Y qué pinta la variable *atraso*? Bueno, es una variable que si te fijas sirve para que cuando un enemigo choque con el protagonista no le quite en un santiamén la energía que tiene,porque normalmente los choques no son instantáneos, sino que el choque se produce durante bastantes frames seguidos, por lo que si en cada frame se le quitara un punto de vida, por cada leve roce con un enemigo nuestro protagonista moriría. El truco que se ha hecho servir es hacer que solamente una vez de cada 50 –cuando *atraso* valga 0,y eso ocurrirá cada 50 frames- se produzca la disminución de energía. Esto querrá decir que en cada choque, la disminución de energía será más pausada y el jugador tendrá más posibilidades de reaccionar y salir del choque sin perder tanta energía.

Además, deberemos añadir una última línea en el Case 1, diciéndole que resetee la variable *health* cuando se reinicie el juego:

```
health=14;
```

Y tendremos que cambiar los Case 2,3 y 4 para volver a la pantalla inicial cuando el personaje muera. Para ello, en los tres Cases, justo después del bloque *if(key(_i))/end*, escribiremos:

```
If (Health==0)
            write (select_fnt,150,100,4,"Game Over:");
            let_me_alone();
            fade(100,0,0,5);
            level=1;
            frame(5000);
            break;
end
```

Aquí hemos utilizado la función **fade()**. Recuerda que esta función activa una transición de colores automática, que puede usarse para oscurecer la pantalla, hacer un fundido, o colorear su contenido. Tiene cuatro parámetros: los tres primeros representan un porcentaje de las componentes primarias del color: R, G y B y el cuarto parámetro es la velocidad: que es un valor entero entre 1 y 64. En el caso concreto de nuestro juego, realizamos un fundido a rojo mate de forma lenta. (este efecto sange mostrará visualmente que hemos muerto)

¡Y ya está¡¿Ya está? ¡Pero los enemigos no salen por ningún lado!Claro, los tenemos que crear. Modifica el Case 2 –el mapa verde- para que tus enemigos se muestren. Yo he creado tres y los he puesto en el centro del mundo, así que tendrás que ir a buscarlos si te quieres encontrar con ellos. Así pues, justo después de crear nuestro protagonista llamando al proceso "MyCharacter()" –dentro del bloque *if(to_item_select_screen)/end*, crearemos tres enemigos, así:

```
enemy_pointer(1600,1400);
enemy_pointer(1750,1300);
enemy_pointer(1200,1750);
```

Ten en cuenta que si lo hacemos así, cada vez que entremos en una casa y volvamos a salir, se volverán a crear estos tres enemigos otra vez –volvemos al Case 2-. Cambia como tu veas si no quieres este comportamiento.

Chequea tu programa ahora. Deberías encontrar algunos enemigos en la pantalla; los podrás matar con tu ataque de espada, pero igualmente, ellos podrán lentamente quitarte la vida, si les tocas.

Ya hemos acabado nuestro pequeño RPG.

## *Concepto y utilidad de los "tiles"

Seguramente habrás apreciado que en el inicio del juego parece que éste se demora un poco: tarda en salir la pantalla inicial.Esto es por una razón clara: necesita cargar en memoria todo el archivo FPG de nuestro juego, el cual contiene entre otras cosas dos grandes mapas de 3000x3000 (el mapa visible y el de durezas). Evidentemente, el tener grandes mapas ocupa muchos recursos al ordenador, porque éste ha de estar permanentemente controlando que el espacio de memoria reservado a estas grandes imágenes permanezca disponible.

En seguida verás que si tenemos más mapas de estas dimensiones, el tamaño del archivo FPG llegará a ser inabarcable y nuestro juego se resentirá en velocidad y posibles cuelgues por no poder llegar a gestionar de golpe tantos píxeles cargados en memoria.

Podríamos pensar en no cargar todos los gráficos al principio del juego. Es decir, hacer diferentes FPGs y cargarlos según fuera necesario. La idea es buena, pero el problema con los mapas grandes los seguiríamos teniendo, y además, en el momento de cargarlos –en mitad del juego- se notaría más el "parón".

Una posibles solución podria ser,por ejemplo, reducir el tamaño del mapa de durezas a la mitad ( o a la cuarta parte,etc)en sus dos coordenadas. Se esta manera, tendríamos un mapa de, por ejemplo 1500x1500, con lo que ganaríamos algo. Evidentemente, si hacemos esto, entonces en todas las líneas de nuestro código donde utilicemos la función **map_get_pixel()** para comprobar durezas, las coordenadas pasadas como parámetros deberían de dividirse por dos ( o por cuatro,etc).

La mejor solución para estos problemas son los "tiles" ("baldosa" en inglés). Un escenario "tileado" es un escenario donde se usan muchas piezas pequeñas que encajan entre sí en vez de una única imagen grande. Por ejemplo, en vez de hacer un gráfico enorme de campo verde con dos árboles se prepara un cuadrado verde y un cuadrado con un árbol y con eso se monta la imagen en tiempo de ejecución. De esta forma ahorras mucha memoria, ya que necesitas

tener menos tamaño total de gráficos cargados, y haces el juego más ligero en lo que a uso de CPU se refiere, ya que no hay nada fuera de pantalla (se pinta sólo lo que queda dentro de la pantalla).Además, si el decorado a "tilear" es un poco repetitivo, se puede ahorrar mucha memoria –piensa en un juego de plataformas 2D- ya que si te fijas verás que los decorados son una repetición contínua de unos pocos tipos diferentes de baldosas: es mucho más económico dibujar 20 baldosas distintas y decirle al programa dónde colocarlas que guardar enormes imágenes de cada fase, que ocuparían mucho. A cambio es un poco más complicado de programar, ya que hay que controlar todas esas pequeñas piezas en vez de una grande, pero eso pasa siempre que se optimiza algo.

Una utilidad evidente para los "tiles" es emplearlos para simular un scroll, por ejemplo, en un RPG. Es decir, se construyen pequeños "cuadraditos" que juntos formarán el fondo e se mueven según convenga, simulando el scroll. Asi se puede conseguir mas rendimiento que con las funciones de scroll de Bennu, que aunque éstas son mas sencillas de programar.

No obstante, es éste un tema relativamente avanzado (y complejo) que requeriría por sí mismo un capítulo entero exclusivamente dedicado a éste. Es por esto que no profundizaremos más en este tutorial básico, y reservamos la explicación de su uso y programación a un apartado del último capítulo de este manual. Remito al lector allí si desea aprender más sobre "tiles".

## *Includes

Otra cosa que habrás descubierto en este tutorial es que cuando los códigos fuente empiezan a sobrepasar cierta longitud, se hacen mucho menos manejables: se pueden cometer más errores, los cambios son engorrosos, se pierde la referencia de lo que se está haciendo…Que el código sea más o menos corto no depende de nosotros: es el que es. Pero lo que sí depende de nosotros es la claridad y pulcritud con la que lo escribamos.

Existe una manera interesante de estructurar y clasificar el código fuente de un juego cuando éste empieza a tener una longitud considerable. Se trata de dividir el código fuente entre varios ficheros de texto diferentes, de manera que cada uno de estos ficheros contenga código sobre partes diferenciadas del total. Por ejemplo, se podría utilizar un fichero para escribir el código del programa principal, otro para escribir los procesos relacionados con el protagonista, otro para escribir los procesos relacionados con los enemigos,etc.De esta manera, se tienen separados los trozos de código por significado o funcionalidad, haciendo mucho más comodo el trabajo de desarrollo y depuración del código. Este proceso es lo que se llama modularización del código.

Y este sistema tiene otra ventaja: podemos reunir en un archivo una serie de procesos o funciones (y secciones con las variables globales,locales,privadas...que utilicen) que sepamos que vamos a reutilizar en varios proyectos, de tal manera que no tengamos que reescribir código ya escrito anteriormente y sólo tengamos que hacer una referencia a ese archivo que contiene el código que nos interesa. Es evidente, pues, que podríamos crearnos nuestras propios almacenes de código listo para ser utilizados desde los más diversos proyectos (unas "librerías" escritas en Bennu mismo),con el ahorro de tiempo y trabajo que eso supone.

Y ¿cómo hacemos referencia dentro de un código fuente "A" a otro archivo el cual contendrá el código "B" que queremos utilizar? Con el comando **include**.

Este comando inserta el contenido de un fichero de código fuente especificado dentro del actual, allí donde el comando esté situado. Tiene un único parámetro que es la ruta del fichero de código fuente a incluir. Dicho de otra manera: este comando hace que el compilador lea el contenido textual de otro fichero (cuyo nombre se indica entre comillas justo después de la palabra INCLUDE) ,lo copie y lo pegue literalmente en la posición donde se encuentra el comando, de tal manera que parezca que se haya escrito todo en el mismo archivo.

El fichero incluido puede tener cualquier extensión, pero se recomienda PRG, ya que no deja de ser un trozo de código Bennu..

Este comando puede usarse en cualquier lugar del código, aunque normalmente se acostumbran a escribir al principio de todo del código ,justo después de las líneas IMPORT.

En realidad, técnicamente INCLUDE no es un comando, es una directiva. Esto a nuestro nivel no nos importa salvo en un par de detallas muy importantes:

1º) El parámetro que tiene NO ha de ir entre parántesis
2º) El punto y coma al final (;) es opcional: puede llevarlo o no.

Conociendo la existencia de este comando, ahora podríamos estructurar el código de nuestros juegos más extensos de forma más ordenada y coherente. Incluso podríamos llegar al extremo de hacer que el fichero Prg principal del juego constara simplemente de una serie de INCLUDES con todos los ficheros secundarios, y el proceso principal.

Como ejemplo muy sencillo de utilización de la orden **include**, aquí teneis dos códigos fuentes. El primero será el que se incluirá dentro del segundo. El primero definirá la acción que realiza una función creada por nosotros, y el segundo la utilizará. Con este ejemplo se pretende mostrar que si quisiéramos, podríamos crear otro programa que hiciera uso también de la función definida en el código primero, de tal manera que el código de ésta sólo hubiera que escribirlo una sóla vez y pudiera ser utilizado múltiples veces.

*Código primero, guardado en el archivo "hola.prg":

```
Function mifuncion( int pepe, int pepa)
Begin
          Return pepe*pepa;
End
```

*Código segundo, guardado en el archivo "miprograma.prg":

```
Import "mod_text";
Import "mod_key";

Include "hola.prg";

Process Main()
Private
          Int mivar;
end
Begin
          Mivar=mifuncion(3,4);
          Write(0,100,100,4,mivar);
          Loop
             if(key(_esc)) break; end
                    Frame;
          end
End
```

Cuando queramos ejecutar "miprograma.prg", no se notará que el cálculo de la multiplicación lo realiza otro fichero aparte. Evidentemente, el fichero "hola.prg" no podrá ser ejecutado sólo por sí mismo.

Otro ejemplo donde se demuestra que **include** se puede escribir en cualquier sitio del programa y leerse varias veces si es necesario, puede ser el siguiente:

*Código primero, guardado en el archivo "hola.prg":

```
if(i<5)
          c = i+b*b;
          elseif(i<7)
                    c = i+b*a;
else
          c = i+a*a;
end
```

*Código segundo, guardado en el archivo “miprograma.prg”:

```
Import “mod_text”;
Import “mod_key”;

Process main()
Private
/*Las variables a,b y c se hacen servir en "hola.prg". También se podrían haber declarado en una sección Private dentro de éste.*/
        Int i,a=2,b=3,c;
end
Begin
        for(i=0;i<10;i++)
                                include "hola.prg"
                                write(0,100,10+10*i,4,c);
        end
        Loop
          if(key(_esc)) break; end
                        Frame;
        end
End
```

Quiero recalcar un detalle. No es lo mismo incluir (con **include**) que importar (con **import**). Con **include** insertamos código fuente dentro de otro código fuente: copiamos texto dentro de otro texto. En cambio, con **import** lo que hacemos es incrustar código binario ya precompilado dentro del código binario generado por nuestro juego, que no es lo mismo.

Con esto que hemos comentado, no sería mala idea coger ahora el código de nuestro RPG –que sobrepasa las 1100 líneas- e irlo diseccionando en diferentes archivos PRG según el significado o ámbito de los diferentes procesos, para hacer más cómoda y rápida la lectura de su código, facilitando así la búsqueda de errores y su mantenimiento.

No hace falta ni decir que esta separación en distintos “módulos” ha de venir de una reflexión y estudio previo sobre los procesos necesarios a implementar en nuestro programa, los tipos de datos a utilizar y las distintas maneras de interaccionar con ellos. De nada sirve empezar a codificar si no se ha hecho una planificación donde se tengan claros los elementos de los que constará nuestro programa.

# CAPITULO 10

## Órdenes y estructuras avanzadas de Bennu

### *Operadores avanzados del lenguaje: el operador ? y los operadores "bitwise"

Supongamos que queremos que una variable de cadena llamada *a* valga "pepito" siempre y cuando la condición *b==1* sea verdadera (*b* sería otra variable, entera en este caso) y queremos que *a* valga "manolito" siempre y cuando dicha condición anterior sea falsa. Para conseguir esto, podremos utilizar el operador "?". En general, si una variable de nuestro programa, dependiendo de si una condición determinada es verdadera o falsa, queremos que tenga un valor u otro diferente respectivamente, estamos en la situación de poder utilizar dicho operador "?".

Es bastante evidente que el operador "?" no es imprescindible, porque su funcionalidad la puede realizar perfectamente un bloque IF/ELSE. Por ejemplo, en el caso en el párrafo anterior, dicha estructura sería así de simple:

```
if(b==1)
    a="pepito";
else
    a="manolito";
end
```

pero "?" ofrece una manera alternativa mucho más compacta, práctica y cómoda a la hora de escribir este tipo de condiciones, sin necesidad de escribir toda una sección IF/ELSE de varias líneas. De todas maneras, al final se trata siempre de una elección personal: existen programadores que prefieren escribir bloques IF/ELSE y otros que en los casos que es posible, utilizan "?".

El funcionamiento del operador "?" es bastante sencillo:

**variable = (condición) ? valor_si_la_condición_es_verdadera : valor_si_la_condición_es_falsa ;**

Vemos que la línea anterior no deja de ser una asignación de un valor a *variable*, pero ese valor puede ser uno u otro dependiendo de si la condición es verdadera o no. La condición a avaluar se escribe con las misma sintaxis y las mismas opciones que las condiciones estudiadas con el bloque IF. Fijarse que entre el valor asignado si la condición es verdadera y el valor asignado si la condición es falsa se escribe el signo **:** . La línea entera finalizará, como viene siendo habitual, con un punto y coma.

Un caso expuesto en el primer párrafo, se escribiría de la siguiente manera con el operador "?"

```
Import "mod_text";
Import "mod_key";

Process Main()
Private
        string a;
        int b=1;
End
Begin
        a= (b==1) ? "pepito" : "manolito";
        write(0,0,0,0,a);
        repeat
              Frame;
        until (key(_esc))
End
```

Otro ejemplo. Supongamos que tenemos una variable contadora *count* que queremos que aumente de valor (en una unidad cada frame) solamente si una variable auxiliar *aux* es mayor de 5 -en el caso de apretar la tecla ENTER-; en caso contrario, el valor de *count* no deberá cambiar.

```
Import "mod_text";
Import "mod_key";

Process Main()
Private
        int aux;
        int count;
End
Begin
        write_var(0,0,0,0,count);
        repeat
              if (key(_enter)) aux=10; end
                count= (aux > 5) ? count+1 : count;
                frame;
        until (key(_esc))
End
```

Otra manera alternativa de haber escrito la línea del operador "?" del ejemplo anterior podría haber sido también ésta:

```
count +=(aux > 5) ? 1 : 0;
```

No es imprescindible que exista una asignación a una variable para utilizar "?"; también se puede utilizar a modo de un IF muy simple. Tan simple que solamente puede incluir una sentencia a ejecutar si la condición es verdadera (aunque ésta podría ser perfectamente una función) y otra si es falsa. Por ejemplo:

```
Import "mod_text";
Import "mod_key";

Process Main()
Private
        int mivar1=1;
End
Begin
(mivar1 < 1 or mivar1 > 1) ? write(0,0,0,0,"Diferente de 1"): write(0,0,0,0,"Igual a 1");
        repeat
              Frame;
        until(key(_esc))
End
```

A parte de este nuevo operador "?" que acabamos de ver, de los operadores aritméticos ya conocidos (+,-,*,/,%...) y de los operadores lógicos (&&, ||, ...), Bennu dispone de una serie de operadores avanzados para trabajar con números conocidos genéricamente como operadores a nivel de bit (ó "bitwise"). Son seis: BAND (&), BOR (|) ,BXOR (^) , BNOT (~), desplazamiento a la derecha (>>) y desplazamiento a la izquierda (<<).

Se llaman así, "a nivel de bit" porque actúan bit a bit. Es decir, trabajan con la representación binaria de los números y operan con cada bit que la forman, siguiendo unas determinadas reglas. Lógicamente, estas reglas sólo están definidas para realizar operaciones sólo con 1 y con 0. Por ejemplo, una operación como ésta: 34 BAND 7 en realidad no tiene mucho sentido: lo más lógico seria escribir esta operación en representación binaria:

```
    100010
BAND   111
      ------
    ??????
```

Las reglas de las seis operaciones son las siguientes:

| **BAND** | **BOR** |
|---|---|
| (Ambos operandos deben ser 1 para que el resultado sea 1) | (Si cualquiera de los dos operandos es 1,el resultado es 1) |

| | |
|---|---|
| 0 & 0 = 0<br>0 & 1 = 0<br>1 & 0 = 0<br>1 & 1 = 1 | 0 \| 0 = 0<br>0 \| 1 = 1<br>1 \| 0 = 1<br>1 \| 1 = 1 |
| **BXOR**<br>(BOR exclusivo: si un elemento o el otro, nunca ambos juntos, son 1, el resultado es 1)<br><br>0 ^ 0 = 0<br>0 ^ 1 = 1<br>1 ^ 0 = 1<br>1 ^ 1 = 0 | **BNOT**<br>(Negado)<br><br>~0 = 1<br>~1 = 0 |
| **Desplazamiento a la derecha**<br>(Mueve hacia la derecha los bits que forman el primer operando un número de posiciones dado por el segundo operando)<br><br>Por ejemplo, 01000101 >> 3 (69>>3) moverá el número 01000101 tres posiciones hacia la derecha, (rellenando los huecos que aparezcan por la izquierda con 0 y destruyendo el valor de los bits que sobrepasan el extremo derecho), resultando el número 00001000.O sea, el 8. | **Desplazamiento a la izquierda**<br>(Mueve hacia la izquierda los bits que forman el primer operando un número de posiciones dado por el segundo operando)<br><br>Por ejemplo, 01000101 << 3 (69<<3) moverá el número 01000101 tres posiciones hacia la izquierda, (rellenando los huecos que aparezcan por la derecha con 0 y destruyendo el valor de los bits que sobrepasan el extremo izquierdo, el cual en el caso de los INT está en 32 posiciones, en el caso de los BYTE está en 8,etc), resultando el número 01000101000. O sea, el 552. |

Así pues, sabiendo esto, la operación de antes, 34 & 7 da como resultado 2 (10 en binario). Podemos comprobar cómo funcionan efectivamente estos operadores, con un sencillito programa:

```
import "mod_text";
import "mod_key";

process main()
begin
        write(0,100,100,4,1 band 2);
        write(0,100,110,4,1 bor 2);
        write(0,100,120,4,1 bxor 2);
        write(0,100,130,4,~1);
        write(0,100,140,4,69>>3);
        write(0,100,150,4,69<<3);
        while(!key(_esc))
                frame;
        end
end
```

Obtenemos como resultado 0,3,3,-2,8,552. ¿Tiene sentido? Los dos últimos valores ya sabemos que sí. Veamos los cuatro primeros:

```
   1 = 0001
& 2 = 0010
  ------------
       0000, o sea, 0

   1 = 0001
| 2 = 0010
  ------------
       0011, o sea, 3

   1 = 0001
```

^ 2 = 0010
---
0011, o sea, 3

~1 = ~...0001=...1110. Hay que tener en cuenta que al cambiar el bit de más a la izquierda por un 1, se entra en el terreno de los números negativos.

Un uso muy común para los operadores a nivel de bits es para hacer flags. Supongamos que tenemos que controlar que un protagonista tiene o no un determinado objeto de entre cinco posibles. Podríamos hacer un array de cinco elementos de tipo byte, por ejemplo, que valiera cada uno 1 si tiene el objeto o 0 si no lo tiene. Pero sabemos que cada byte tiene 256 valores posibles, así que usando sólo dos (0 y 1 ) desperdiciamos 254 posibilidades para cada byte (y si hay cinco bytes, pues multiplica). Se ve por tanto que es una solución bastante despilfarradora. No obstante, ya sabes que cada byte se compone de 8 bits, así que cogiendo cinco bits de un byte, ya podríamos tener para cada uno de ellos los dos valores (0 y 1) que precisamente necesitamos para cada objeto. Es decir, en un solo byte podemos obtener información de hasta ocho valores binarios independientes.

De hecho, si a un byte le hacemos un BAND con el valor 00000001, podemos descartar los bits que no nos interesan y conocer el valor del único bit que queremos saber, que en este caso sería el de más a la derecha. Es decir: sea un byte xxxxxxxx y le hacemos un BAND con 00000001 (que llamaremos máscara), los primeros 7 bits siempre darán cero, valga lo que valga nuestro primer dato, pero el ultimo valdrá cero si el ultimo bit es un cero, o uno si el ultimo bit es un uno. De esta forma, si usando nuestra máscara nos esta operación nos da un valor cero, es que el protagonista no tiene el objeto correspondiente al bit de más a la derecha, y si nos da un uno, sí lo tiene. Si queremos saber lo mismo para el objeto asociado al segundo bit de más a la derecha, deberíamos hacer un BAND del byte xxxxxxxx con el valor 00000010. Y así con los demás bites.

## *Configuración de la orientación del gráfico de un proceso con la variable XGRAPH

Supongamos que tenemos la necesidad de que dependiendo del valor de la variable ***ANGLE*** de un proceso (es decir, de la orientación de su gráfico), dicho gráfico varíe. Un caso típico son los RPGs de vista zenital (vista desde arriba), donde dependiendo de si el proceso viaja al norte, este, oeste ó sur el gráfico será diferente, ya que estará encarado a la dirección donde viaja. En anteriores capítulos se han visto ejemplos de esta situación; la manera más directa de solventarla es haciendo uso de un SWITCH, así, por ejemplo:

```
Switch(angle)
	Case 0..23000 or 338000..360000:
	//DERECHA. Aquí se cambiaría la variable GRAPH para mostrar el gráfico correcto
	End
	Case 23000..68000://(45 grados mas)
	//DERECHA-ARRIBA. Ídem
	End
	Case 68000..113000:
	//ARRIBA
	End
	Case 123000..158000:
	//IZQUIERDA-ARRIBA
	End
	Case 158000..203000:
	//IZQUIERDA
	End
	Case 203000..248000:
	//IZQUIERDA-ABAJO
	End
	Case 248000..293000:
	//ABAJO
	End
	Case 293000..338000:
	//DERECHA-ABAJO
	End
End
```

De forma similar, si quisiéramos que el personaje tuviera un gráfico diferente dependiendo de la dirección a la que se dirije, dada por un clic de ratón, se podría hacer algo así (es la misma idea):

```
If(tipico_id.x<mouse_posx)
       //DERECHA. Aquí se cambiaría la variable GRAPH para mostrar el gráfico correcto
End
If(tipico_id.x>mouse_posx)
       //IZQUIERDA. Ídem
End
If(tipico_id.y<mouse_posy)
       //ABAJO
End
If(tipico_id.y>mouse_posy)
       //ARRIBA
End
```

Otro ejemplo ingenioso para solventar una problemática similar es el código siguiente. Requerirá no obstante que dispongas de un FPG llamado "graficos.fpg" formado por 20 imágenes: las primeras 5 corresponderán al gráfico del proceso dirigiéndose hacia abajo (son 5 porque se realiza una animación para crear el efecto de que el gráfico "camina"); las 5 siguientes corresponderán al gráfico del proceso mirando hacia la izquierda, los 5 siguientes serán mirando hacia arriba y los 5 últimos mirarán hacia la derecha.

```
import "mod_text";
import "mod_key";
import "mod_proc";
import "mod_map";

GLOBAL
  int grafico[4][5];
  int direccion;
  int animacion;
  int contador=1;
  int i, j;
END

PROCESS MAIN()
BEGIN
  load_fpg("graficos.fpg");
/* Con esto relleno el array de gráficos, con los correspondientes codigos de dentro del
FPG. Están en orden: los primeros 5 son para la direccion 1, los siguientes para la
direccion 2, consecutivamente.*/
  FOR(i=1;i<=4;i++)
          FOR(j=1;j<=5;j++)
                    grafico[i][j]=contador;
                    contador++;
          END
  END
  proceso();
  REPEAT
             FRAME;
  UNTIL(key(_esc))
  let_me_alone();
END


PROCESS proceso()
BEGIN
   x=160; y=120;
   direccion=1;
   write_var(0,270,10,0,direccion);
   LOOP
          //Sólo 4 direcciones posibles
          if(key(_left) AND direccion<4) direccion++; end
          if(key(_right) AND direccion>1) direccion--; end

          //Con animacion hacemos el bucle con las 5 imagenes que tiene la animación
          animacion++;
```

```
            IF(animacion>5) animacion=1; END

/*Esta es la linea importante.Si direccion vale 1, mira hacia abajo, si 2 a la
derecha... La variable animacion va siempre del 1 al 5 y mostrará la postura
que toque en cada momento. Aunque cambie de dirección, si está apoyando el pie
izquierdo, seguirá con ese pie.*/
            graph=grafico[direccion][animacion];
            FRAME;
    END
END
```

No obstante, tenemos otra manera de hacer todo esto "más profesional", que es utilizando la variable local ***XGRAPH*** , definida en el módulo "mod_grproc" (en combinación con ***ANGLE***).

El valor de esta variable ha de ser la dirección de memoria de una tabla de variables de tipo INT (es decir, el nombre de esa tabla precedido de "&"),de manera que el efecto de la variable ***ANGLE*** no sea el de rotar el gráfico, sino el de seleccionar un elemento de esta tabla. Es decir, esta tabla básicamente contendrá los identificadores de los gráficos asociados al proceso en cuestión, que se visualizarán en función de su ángulo. No obstante,el primer elemento de esta tabla debe ser siempre el número de gráficos que van a utilizarse (el número total de elementos de la tabla, menos uno) y ya a partir del segundo elemento elemento de la tabla se podrán especificar la serie de identificadores de gráfico válidos. El valor por defecto de esta variable es 0.

El intérprete Bennu dividirá internamente los 360 grados que corresponden a la rotación completa del gráfico entre los gráficos disponibles, de forma homogénea (el ángulo 0 siempre para el primer gráfico), eligiendo en cada frame el gráfico de la tabla adecuado en función del valor de ***ANGLE***. Un valor negativo en esta tabla indica que el gráfico en cuestión se mostrará invertido horizontalmente. Por ejemplo, un valor de -2 sirve para especificar el gráfico de código 2, reflejado horizontalmente.

El ejemplo siguiente es un código sencillo de muestra, bastante autoexplicativo:

```
import "mod_key";
import "mod_proc";
import "mod_map";
import "mod_grproc";

global
        int tabla[5];
end

process main()
private
            int graf1,graf2,graf3,graf4,graf5;
end
begin
  x=160;y=120;
//Genero 5 gráficos que se van a ver dependiendo de la orientación del proceso
  graf1=new_map(30,30,32);map_clear(0,graf1,rgb(255,0,0));
  graf2=new_map(30,30,32);map_clear(0,graf2,rgb(0,255,0));
  graf3=new_map(30,30,32);map_clear(0,graf3,rgb(0,0,255));
  graf4=new_map(30,30,32);map_clear(0,graf4,rgb(255,255,0));
  graf5=new_map(30,30,32);map_clear(0,graf5,rgb(255,0,255));

//Relleno la tabla cuya dirección de memoria se la pasará a Xgraph
  tabla[0]=5;     //Número de gráficos a emplear
  tabla[1]=graf1; //Id del gráfico que se mostrará para los angles más pequeños
  tabla[2]=graf2; //El siguiente gráfico, y así...
  tabla[3]=graf3;
  tabla[4]=graf4;
  tabla[5]=graf5;

  proceso();
end

process proceso()
begin
//Con esta línea, el significado de ANGLE pasa a ser totalmente diferente
```

```
  xgraph=&tabla;
  repeat
        if(key(_up)) angle=(angle+5000)%360000; end
        if(key(_down)) angle=(angle-5000)%360000;end
        frame;
  until(key(_esc))
end
```

## *La estructura local predefinida RESERVED

La estructura local predefinida RESERVED contiene información interna de Bennu. Sin embargo, en alguna ocasión es posible que se quieran utilizar algunos campos de ella. Utilizar en el sentido solamente de leer sus valores, ya que bajo ninguna circunstancia es buena idea alterar sus valores si no se sabe lo que se está haciendo.

Si algunos campos de esta estructura no están documentado en la Wiki significa que su posible uso es bastante limitado. Deberías consultar el código fuente de Bennu para saber cuál es el sentido de dichos campos, además de tener un nivel de programación en C ciertamente elevado.

Algunos de los campos más importantes de esta estructura para los programadores de Bennu son:

<table>
<tr><td>RESERVED.PROCESS_TYPE</td><td>Es de tipo INT</td><td>Identificador de tipo de proceso. Es un número siempre menor de 65537.</td></tr>
<tr><td>RESERVED.STATUS</td><td>Es de tipo INT</td><td>Estado del proceso. Puede contener los siguientes valores:<br><table>
<tr><td>STATUS_DEAD</td><td>Igual a 0</td><td>El proceso está muerto</td></tr>
<tr><td>STATUS_KILLED</td><td>Igual a 1</td><td>El proceso está killed</td></tr>
<tr><td>STATUS_RUNNING</td><td>Igual a 2</td><td>El proceso está ejecutándose</td></tr>
<tr><td>STATUS_SLEEPING</td><td>Igual a 3</td><td>El proceso está dormido</td></tr>
<tr><td>STATUS_FROZEN</td><td>Igual a 4</td><td>El proceso está congelado</td></tr>
<tr><td>STATUS_WAITING</td><td>Igual a 7</td><td>El proceso está en espera</td></tr>
</table></td></tr>
<tr><td>RESERVED.BOX_X0</td><td>Es de tipo INT</td><td>Coordenada horizontal de la esquina superior izquierda del gráfico del proceso</td></tr>
<tr><td>RESERVED.BOX_Y0</td><td>Es de tipo INT</td><td>Coordenada vertical de la esquina superior izquierda del gráfico del proceso</td></tr>
<tr><td>RESERVED.BOX_X1</td><td>Es de tipo INT</td><td>Coordenada horizontal de la esquina inferior derecha del gráfico del proceso</td></tr>
<tr><td>RESERVED.BOX_Y1</td><td>Es de tipo INT</td><td>Coordenada vertical de la esquina inferior derecha del gráfico del proceso</td></tr>
</table>

Un ejemplo:

```
import "mod_say";

process main()
begin
    p1();
end

process p1()
begin
    p2();
```

```
    p3();
end

process p2()
begin
    p3();
end

process p3()
begin
    if (father.reserved.process_type == type p1)
        say ("Mi padre es p1");
    else
        say ("Mi padre no es p1");
    end
end
```

***Configuración del escalado con las variables globales SCALE_MODE y SCALE_RESOLUTION. Orden "get_modes()"**

Existe una variable global predefinida llamada ***SCALE_MODE***, cuyo valor es recomendable asignarlo (si es que se va a asignar, cosa que es totalmente opcional) siempre antes de llamar a la función **set_mode()**.Esta variable define el modo de escalado de los gráficos en pantalla y sus posibles valores son:

| | | |
|---|---|---|
| **SCALE_NONE** | Equivale a 0 | Desactiva el escalado (valor por defecto) |
| **SCALE_SCALE2X** | Equivale a 1 | Actual filtro de escalado 2x (realiza un suavizado rápido en los contornos de la imagen, sin excesiva calidad) . También se puede activar usando el valor correspondiente ya conocido del cuarto parámetro de **set_mode()** estableciendo entonces ***SCALE_MODE*** a SCALE_NONE |
| **SCALE_HQ2X** | Equivale a 2 | Escalado 2x, con filtro hq2x (realiza un suavizado de muy alta calidad en los contornos de la imagen, a cambio de un menor rendimiento) |
| **SCALE_SCANLINE2X** | Equivale a 3 | Escalado 2x, con filtro scanline (intercala una línea negra entre cada línea de dibujo) |
| **SCALE_NOFILTER ó SCALE_NORMAL2X** | Equivale a 4 | Escalado 2x, sin filtro (redimensiona cada pixel a 2x2 sin ningún tipo de suavizado) |

Un ejemplo mostrando el efecto de su uso será lo más fácil para comprender su funcionalidad:

```
Import "mod_video";
Import "mod_text";
Import "mod_key";
Import "mod_map";
Import "mod_proc";
Import "mod_draw";

Process Main()
begin
  set_mode(320,240,16);
  write(0,100,50,0,"Valor de SCALE_MODE:");
  write_var(0,250,50,0,scale_mode);
  proceso();
  repeat
          if(key(_a)) scale_mode=SCALE_NONE;set_mode(320,240,16);end
          if(key(_s)) scale_mode=SCALE_SCALE2X;set_mode(320,240,16);end
          if(key(_d)) scale_mode=SCALE_HQ2X;set_mode(320,240,16);end
          if(key(_f)) scale_mode=SCALE_SCANLINE2X;set_mode(320,240,16);end
          if(key(_g)) scale_mode=SCALE_NOFILTER;set_mode(320,240,16);end
          frame;
  until(key(_esc))
```

```
  let_me_alone();
end

Process proceso()
begin
  x=180;y=100;
  graph=new_map(60,60,16);
  drawing_map(0,graph);
  drawing_color(rgb(255,255,0));
  draw_fcircle(25,25,25);
  loop
          frame;
  end
end
```

Por otro lado, también existe la variable global predefinida ***SCALE_RESOLUTION***, la cual sirve para escalar un modo de video "hacia abajo", a una escala menor. Es decir, el modo interno para el juego seguirá siendo el especificado con **set_mode()**, pero el modo a dibujar será el especificado en ***SCALE_RESOLUTION***. La mayor utilidad de esta variable está en la facilidad que ofrece para poder portar juegos a consolas portátiles que no permitan las resoluciones que permite un PC estándar; de esta manera, se puede programar en un ordenador un videojuego pensado para hacerlo funcionar en una consola sin preocuparse por reescalar a mano los gráficos.

Su uso es de la siguiente manera:

```
scale_resolution = WWWWHHHH ;
```

donde "WWWW" es el ancho y "HHHH" es el alto. Éstos pueden ser cualquier valor menor que la resolución especificada por **set_mode()**

Un ejemplo:

```
Import "mod_video";
Import "mod_text";
Import "mod_key";
Import "mod_map";
Import "mod_proc";
Import "mod_draw";

Process Main()
begin
  set_mode(1024,768);
  write(0,100,50,0,"Valor de SCALE_RESOLUTION:");
  write_var(0,350,50,0,scale_resolution);
  proceso();
  repeat
          if(key(_a)) scale_resolution=08000600;end
          if(key(_s)) scale_resolution=06400480;end
          if(key(_d)) scale_resolution=03200240;end
          if(key(_f)) scale_resolution=10240768;end
          frame;
  until(key(_esc))
  let_me_alone();
end

Process proceso()
begin
  x=180;y=100;
  graph=new_map(60,60,32);
  drawing_map(0,graph);
  drawing_color(rgb(255,255,0));
  draw_fcircle(25,25,25);
  loop
          frame;
  end
end
```

Es interesante conocer también la existencia de la función **get_modes()**, definida en el módulo "mod_video". Esta función tiene dos parámetros, que se corresponden con los dos últimos parámetros de **set_mode()**; es decir, la profundidad de vídeo (8,16,32 bits) y los distintos "flags" (fullscreen, modal, doublebuffer,etc), y devuelve la dirección de memoria de una variable entera (es decir, un puntero a INT, o dicho de otra manera, una variable de tipo INT POINTER). Este valor devuelto contendrá la lista de los diferentes modos de vídeo soportados, para la profundidad y flags especificados, por el hardware que esté ejecutando en ese mismo momento el programa. Haciendo uso, pues, de esta función, tendremos el conocimiento de forma anticipada sobre qué modos de vídeo podemos utilizar y cuáles no, y hacer que el programa reaccione en consecuencia. Si no se encuentra ningún modo de vídeo disponible, **get_modes()** devolverá 0, y si se encuentran todos los modos de vídeo que es capaz de soportar Bennu disponibles en ese hardware, **get_modes()** devolverá -1. Un ejemplo:

```
import "mod_say";
import "mod_video";
import "mod_key";

Process Main()
Private
    int pointer modes;
End
Begin

    modes = get_modes(8, MODE_FULLSCREEN);
    say ("Modos 8bit");
    say ("----------");
    if (modes == 0)
        say ("No hay modos de video disponibles");
    elif (modes == -1 )
        say ("Cualquier modo de video está disponible");
    else
        while (*modes!=0)
            say (*modes++ + " x " + *modes++ );
        end
    end
    say ("");

    modes = get_modes(16, MODE_FULLSCREEN);
    say ("Modos 16bit");
    say ("----------");
    if (modes == 0)
        say ("No hay modos de video disponibles");
    elsif (modes == -1 )
        say ("Cualquier modo de video está disponible");
    else
        while (*modes!=0)
            say (*modes++ + " x " + *modes++ );
        end
    end
    say ("");

    modes = get_modes(24, MODE_FULLSCREEN);
    say ("Modos 24bit");
    say ("----------");
    if (!modes)
        say ("No hay modos de video disponibles");
    elsif (modes == -1 )
        say ("Cualquier modo de video está disponible");
    else
        while (*modes!=0)
            say (*modes++ + " x " + *modes++ );
        end
    end
    say ("");

    modes = get_modes(32, MODE_FULLSCREEN);
    say ("Modos 32bit");
    say ("----------");
```

```
    if (!modes)
        say ("No hay modos de video disponibles");
    elsif (modes == -1 )
        say ("Cualquier modo de video está disponible");
    else
        while (*modes!=0)
            say (*modes++ + " x " + *modes++ );
        end
    end
repeat frame; until (key(_esc))
End
```

## *Las variables globales predefinidas DUMP_TYPE y RESTORE_TYPE

Los modos “dump” se utilizan para definir el tipo de volcado de imagen necesario a aplicar a la pantalla, asignando el modo deseado a la variable global predefinida ***DUMP_TYPE***. Los modos “dump” influyen en la manera que tiene el gráfico de los procesos en dibujarse sobre la pantalla. Esta variable puede tener los siguientes valores:

| | | |
|---|---|---|
| **PARTIAL_DUMP** | Equivale a 0 | Sólo los gráficos de los procesos actualizados (los que hayan cambiado) serán redibujados en cada frame. Es el valor por defecto. |
| **COMPLETE_DUMP** | Equivale a 1 | Los gráficos de todos los procesos serán redibujados a cada frame, independientemente de que hayan cambiado o no. |

El modo de “dump parcial” es útil si no hay (relativamente) demasiados procesos que cambian, y cuando hay muchos procesos que no necesitan ser actualizados, al contrario que el modo de “dump completo”.

Está claro que dibujar completamente todos los procesos a cada frame consume muchos más recursos de la máquina que no actualizar sólo los gráficos de algunos de ellos. No obstante, el modo de “dump completo” puede ser útil, porque el redibujado parcial de procesos puede causar probablemente efectos indeseados en determinadas situaciones, sobretodo en combinación con determinados modos “restore”. Asegúrate de qué modos “restore” son compatibles con el modo “dump” que tengas activado, ya que, repito, algunas combinaciones pueden causar algunos problemas.Sin embargo, si no experimentas ningún problema, puedes dejar el valor por defecto de ***DUMP_TYPE***

Por otro lado, los modos “restore” se utilizan para definir el tipo de restauración necesaria para aplicar al fondo de la pantalla, asignando el modo deseado a la variable global predefinida ***RESTORE_TYPE***. Los modos “restore” influyen en la manera que tiene el fondo de pantalla en restaurarse en cada frame.

| | | |
|---|---|---|
| **NO_RESTORE** | Equivale a -1 | El fondo no será restaurado (o sea, redibujado) |
| **PARTIAL_RESTORE** | Equivale a 0 | El fondo será restaurado en las áreas donde los gráficos hayan sido pintados de nuevo en el último frame |
| **COMPLETE_RESTORE** | Equivale a 1 | El fondo será restaurado completamente en cada frame. |

El modo de “retore parcial” es útil si no hay (relativamente) demasiados cambios en el fondo, al contrario que el modo de “restore completo” (como sería el caso de los scrolls a pantalla completa parcialmente transparentes).

Está claro que restaurar completamente el fondo de pantalla a cada frame consume muchos más recursos de la máquina que actualizar solamente partes de él. Sin embargo, detectar qué partes necesitan ser actualizadas no es una tarea fácil. Así que pueden existir situaciones en que la restauración total sea más óptima.

Un ejemplo bastante claro de las consecuencia de su uso es el siguiente:

```
import "mod_map";
import "mod_key";
import "mod_video";
import "mod_screen";
import "mod_text";
```

```
import "mod_draw";
import "mod_rand";
import "mod_proc";

process main()
private
   int fondo;
end
begin
    set_mode(320,240,32);
    graph=new_map(30,30,32);map_clear(0,graph,rgb(255,0,0));
    fondo=new_map(320,240,32);map_clear(0,fondo,rgb(0,255,0));
    put_screen(0,fondo);
    bola();
    write_var(0,0,0,0,dump_type);
    write_var(0,0,10,0,restore_type);
    repeat
/*Los valores por defecto de dump_type es partial_dump y de restore_type es
partial_restore*/
        if(key(_a)) dump_type=partial_dump; end
        if(key(_b)) dump_type=complete_dump;end
        if(key(_c)) restore_type=partial_restore; end
        if(key(_d)) restore_type=complete_restore;end
        if(key(_e)) restore_type=no_restore; end

        if(key(_right)) x=x+10; end
        if(key(_left)) x=x-10; end
        if(key(_up)) y=y-10; end
        if(key(_down)) y=y+10; end
        frame;
    until (key(_esc))
    exit();
end

process bola()
begin
    graph=new_map(30,30,32);map_clear(0,graph,rgb(0,0,255));
    loop
          x=rand(0,320);
              y=rand(0,240);
          frame(500);
    end
end
```

## *Trabajar de forma básica con punteros y con gestión dinámica de memoria

Un puntero es un tipo de variable que “apunta” a otra variable. Es decir, su valor es siempre la dirección de memoria del ordenador donde esa otra variable está ubicada.

Los punteros se pueden entender como una forma de ahorrar memoria, a la vez que sirven para conseguir que los programas no tengan límites. Supongamos que tenemos una lista de, por ejemplo, las películas de una tienda. Lógicamente, si queremos trabajar con esta lista en nuestro programa, tendríamos que guardar la información en variables. Supongamos que cada película ocupa una posición de un vector de enteros, por ejemplo. Está claro que en este caso necesitaremos suficientes posiciones para almacenar todas las películas: si hay 20 películas, pues 20 posiciones, usaremos un array de 20 posiciones. No parece haber ningún problema con esto. Pero ¿qué pasa por ejemplo si se venden películas? Las deberíamos eliminar del array. Pero entonces quedarían huecos vacíos, y eso es desperdiciar memoria. Pero hay un caso peor ¿y si en la tienda entran nuevas películas? Ya no hay sitio, no caben más de 20 películas en nuestro vector, por lo que habría que rehacer el programa.

Podríamos pensar que una posible solución a este problema podría ser crear un vector con tantas posiciones como películas existen en el planeta. No obstante, aparte de ser una bestialidad por la cantidad de memoria que desaprovecharíamos, nada te aseguraría de que no se siguieran haciendo películas... Lo bueno de los punteros es

que puedes pedir o ceder memoria en tiempo de ejecución (es decir, cuando se está ejecutando, después de haber sido programado y compilado) y no de compilación (es decir, cuando se está programando el código). En este caso concreto, para añadir películas sin problemas, se podría utilizar un vector de estructuras (llamadas "nodos"), las cuales contendrían por un lado la información referente a una película pero además un puntero que se usaría para "señalar" donde está el siguiente nodo en la memoria. De esta manera, la lista de películas podría aumentar ó disminuir y lo que se estaría haciendo sería añadir ó eliminar (respectivamente) nodos del vector dinámicamente : el vector de nodos podrá tener en un momento determinado un número de nodos y en otro momento otro número de nodos diferente, cada uno de los cuales "apuntaría" donde está el siguiente. Éste es un método realmente eficaz para ahorrar memoria...aunque realmente los punteros no pueden apuntar a toda la memoria existente del ordenador.En Bennu, por ejemplo, se reservan 32 bites para almacenar el valor de un puntero cualquiera, por lo que se pueden direccionar hasta $2^{32}$ direcciones, y por tanto, se pueden ubicar hasta 4GB de memoria haciendo uso de punteros. Que ya es suficiente,creo.

No obstante, estas aplicaciones tan útiles que tienen los punteros no se expondrán en este texto, ya que se salen de su ámbito introductorio. Si se desea profundizar en estos aspectos avanzados de la gestión dinámica de memoria, recomiendo leer libros de programación (en C) que hablen sobre los distintos tipos de listas (simplemente enlazadas, doblemente enlazadas, circulares,etc), pilas, colas, árboles, grafos, etc.

Los punteros tienen otras aplicaciones. Por ejemplo, al indicar direcciones de memoria, pueden acceder a variables de otras funciones, e incluso se podría dar el caso de direcciones de memoria de otro hardware, como tarjetas de sonido, dispositivos usb o cualquier aparato conectado por los puertos serie o paralelo, aunque esto es un asunto más complejo en el que intervienen una serie de elementos que complican las cosas demasiado para explicarse en este texto. De hecho, si los punteros no los entiendes del todo bien o no ves para qué los podrías hacer servir, puedes olvidarlos...pero recuerda al menos que existen, para cuando los necesites.

Hemos dicho que los punteros no son más que otro tipo de variables, que tienen la particularidad de almacenar direcciones de memoria de otras variables. Por lo tanto, como variables que son ellos mismos, deberemos declararlos. Esto se puede hacer de dos maneras equivalentes:

```
tipoDato POINTER nombrePuntero ;
ó
tipoDato * nombrePuntero ;
```

donde "tipoDato" es el tipo de dato (INT,DWORD,FLOAT,STRING,etc) de la variable que será apuntada por el puntero. Es decir (y esto es importante), al declarar un puntero debemos saber de antemano a qué tipo de variable apuntará, porque no es lo mismo que un puntero apunte a una variable Int que a una variable String, por ejemplo. Por eso se ha de indicar el "tipoDato".

Una vez declarado, ya podemos usar el puntero. Lo más común es empezar asignándole un valor. Supongamos que tenemos un puntero ya declarado llamado *punt* (que está pensado para apuntar a una variable de tipo Int, por ejemplo) y deseamos asignarle la dirección de memoria de una variable de tipo Int llamada *a*. Esto se puede hacer de dos maneras:

```
nombrePuntero = & nombreVariable ;
ó
nombrePuntero = OFFSET nombreVariable ;
```

Es decir, fíjate que para indicar la dirección de memoria de la variable *a* se le antepone el símbolo & (o bien la palabra clave "offset"). Es decir, "&a" se puede leer como "dirección de memoria de la variable a". Ojo con no olvidarse de escribir este símbolo, porque por ejemplo, si hiciéramos *punt=a;* , Bennu se quejaría diciendo que hay un error, ya que *punt* sólo puede valer direcciones de memoria, y en esa línea lo que estaríamos haciendo es forzar a que *punt* valiera lo mismo que *a*, que tuviera su mismo contenido, cosa que no puede ser.

Por otro lado, también has de saber que si tenemos un puntero ya apuntando a la dirección de una variable concreta, podemos modificar el contenido de dicha variable utilizando el puntero. Es decir, en el caso de tener *punt* apuntando a *a,* podríamos modificar el valor de *a* utilizando *punt* . Esto se puede hacer de dos maneras:

```
POINTER nombrePuntero = nuevoValorParaLaVariableApuntadaPorElPuntero ;
ó
* nombrePuntero = nuevoValorParaLaVariableApuntadaPorElPuntero ;
```

Veamos todo esto con un ejemplo:

```
import "mod_text";
import "mod_key";

process main()
private
          int myvar1= 777;
          int myvar2[5]=1,2,3,4,5;

          int pointer myp1; //Puntero que apunta a una variable de tipo Int
          int pointer myp2; //Puntero que apunta a una variable de tipo Int
end
begin
  myp1= &myvar1; //Asigno a myp1 la dirección de memoria de la variable entera myvar1
  *myp1= 111; //Modifico el valor de la variable apuntada por myp1 (es decir, myvar1)

  myp2= &myvar2; //Asigno a myp2 la dirección de memoria de la variable myvar2
/*Atención: myvar2 no es una variable entera (se supone que myp2 apunta a variables
enteras).Pero no pasa nada, porque myvar2 es un vector de variables enteras. En este
caso, myp2 apuntará siempre al primer elemento de ese vector, es decir, a myvar2[0]. */
  *myp2=9; //Modifico el valor de myvar2[0]

/*Existe un detalle interesante cuando un puntero apunta a un vector. Y es que se pueden
modificar los valores de otros elementos que no sean el primero de igual manera que se
harían con el propio vector. Es decir, si queremos modificar myvar2[2], podríamos hacer
perfectamente (ahora ya sin símbolo *): */
  myp2[2]= 99;

  //Muestro los contenidos modificados de la variables
  write_var(0,100,100,0, myvar1);
  write_var(0,100,120,0, myvar2[0]);
  write_var(0,100,140,0, myvar2[2]);

  while (!key(_esc))  frame; end
end
```

Otro ejemplo, donde se puede comprobar que el valor que tiene un puntero (recordemos, una dirección de memoria), y el valor que contiene la variable apuntada por éste son cosas diferentes:

```
import "mod_say";

Process Main()
private
    int my_int;
    int* my_int_pointer;
End
Begin
    my_int = 4;
    my_int_pointer = &my_int;
    say("my_int: " + my_int);
/*Muestro el contenido de my_int_pointer (en formato hexadecimal),el cual es la dirección
de memoria de la variable apuntada por él (es decir, my_int). En cada ejecución puede ser
diferente.*/
    say("my_int_pointer: " + my_int_pointer);
/*El símbolo * hace que lo que se muestre es el valor de la variable apuntada por
my_int_pointer (es decir, el valor de my_int, o sea, 4 */
    say("*my_int_pointer: " + *my_int_pointer);
End
```

Los punteros pueden apuntar a cualquier tipo de datos (Ints, Strings, etc) e incluso a tipos de datos definidos por el usuario (TYPEs). Y también pueden ser usados como parámetros de funciones. Veamos un ejemplo:

```
import "mod_say";

Type punto
```

```
    int x;
    int y;
End

Type persona
    string nombre;
    int edad;
End

Process Main()
Private
    int my_int;
    int * my_intpuntero;
    punto myPoint;
    persona Person;
    persona * personPointer; // Posible, porque "persona" es un tipo de datos
    //Person* personPointer; // No posible, porque "Person" no es un tipo de datos
End
Begin
    my_intpuntero = &my_int;
    my_int = 3;
    say(my_int);
/*La línea siguiente Debería mostrar el contenido de la variable apuntada por
my_intpuntero (es decir, el valor de my_int, o sea, 3)*/
    say(*my_intpuntero);

/*Ahora modifico el valor de la variable apuntada por my_intpuntero (es decir, my_int).
Por lo tanto, los dos says siguientes deberán mostrar un 4*/
    *my_intpuntero = 4;
    say(my_int);
    say(*my_intpuntero);

    personPointer = &Person;
/*La línea siguiente modifica el valor de un campo de la estructura apuntada por
personPointer (es decir Person).Realmente la forma "correcta" de escribir esto es
(*personPointer).nombre="Mies", pero se puede escribir así también. Los dos says
siguientes deberán mostrar "Mies"*/
    personPointer.nombre = "Mies";
    say(Person.nombre);
    say(personPointer.nombre);

/*Paso como parámetro la dirección de memoria de la variable myPoint, la cual será
recogida -ver su cabecera- por el puntero p */
    setXY(&myPoint);
/*La función setXY ha modificado los campos de myPoint, con lo que los says siguientes
deberán mostrar lso números 3 y 5, respectivamente.Fijarse que acabamos de hacer que las
funciones puedan devolver más de un valor: los punteros nos pueden servir pues para no
limitarnos a devolver un valor con return, sino devolver los valores que queramos, y del
tipo que queramos*/
    say(myPoint.x);
    say(myPoint.y);

End

Function int setXY(punto * p)
Begin
/*Modificamos los valores de la estructura apuntada por p (es decir myPoint).Realmente la
forma "correcta" de escribir esto es (*p).x=3 y (*p).y=5 , pero se puede escribir así
también*/
    p.x = 3;
    p.y = 5;
End
```

A la hora de trabajar con punteros y ubicaciones de memoria, podemos hacer uso de unas cuantas funciones, conocidas genéricamente como función de gestión dinámica de memoria.

| ALLOC (INT TAMAÑO) |
|---|
| Esta función, definida en el módulo "mod_mem", reserva un bloque de memoria de un determinado tamaño, especificado en bytes, para su posterior uso exclusivo por parte de nuestro programa.<br><br>Esta función puede devolver error (valor -1) si al intentar reservar memoria se encuentra con que no hay suficiente memoria libre disponible para el tamaño especificado. De todas maneras, este error es bastante inusual.<br><br>PARAMETROS:<br>INT TAMAÑO : Número de bytes que ocupará el bloque de memoria a reservar<br><br>VALOR RETORNADO:<br>POINTER : Puntero que apunta al primer elemento (el principio) del bloque de memoria reservada |

| FREE(POINTER PUNTERO) |
|---|
| Esta función, definida en el módulo "mod_mem", libera un bloque de memoria reservado previamente con **alloc()** (o funciones similares,como **calloc()** ó **realloc()** )<br><br>El puntero que se le pasa como parámetro ha de ser el puntero que apunte al dicho bloque de memoria, por lo que normalmente se usará el puntero que contiene la salida que se obtuvo en su momento con **alloc()** o similares.<br><br>PARAMETROS: POINTER PUNTERO: Puntero que apunta al bloque de memoria a liberar |

Cuando se reserva memoria con **alloc()**, la dirección de memoria que esta función devuelve (y que guardamos en un puntero) hemos dicho que se corresponde al principio del bloque de la memoria que se acaba de reservar. Pero tenemos la posibilidad de que dicho puntero "apunte" a cualquier otro sitio de dicho bloque. Es decir, que vaya "avanzando" a través del bloque, o si queremos, podemos hacer que "retroceda" también. Enseguida veremos cómo.

Justamente para saber cómo debemos mover el puntero para adelante o para atrás dentro de nuestro bloque de memoria, debemos tener en cuenta un detalle muy importante: el tipo de datos que se especificó en la declaración del puntero. Por ejemplo, si el puntero apunta a una variable Byte (es decir, su declaración es del tipo *byte pointer mipuntero;* )cada vez que el puntero avance o retroceda, lo hará en un byte. Si el puntero apunta a una variable Word cada vez que éste se mueva lo hará de dos bytes en dos bytes (un word son 2 bytes). Si el puntero apunta a una variable Int, cada vez que éste se mueva lo hará de 4 bytes en 4 bytes (un int son 4 bytes). Y así.

Existen dos maneras equivalentes para avanzar ó retroceder un puntero de un tipo determinado en un bloque de memoria reservado:

| | |
|---|---|
| **nombrePuntero + i** | Sumamos una cantidad entera "i" que representa el número de elementos que el puntero avanzará, a partir del lugar donde esté en ese momento. Dependiendo del tipo de dato al que apunte el puntero, "i" representará el número de bytes, words, int, etc a avanzar.<br>Para retroceder una cantidad "i" determinada de elementos, simplemente hay que escribir **nombrePuntero – i**<br><br>Es tarea del programador procurar que el puntero no sobrepase los límites inferior y superior del bloque de memoria reservada. Si ocurriera esto, los resultados son imprevisibles ya que se estaría apuntando a un lugar de la memoria ocupada por cualquier otra cosa. |
| **nombrePuntero[i]** | Se utiliza el nombre del puntero como si de un vector se tratara, de forma que el elemento "i" de ese "vector" se corresponde con el elemento número "i" dentro del bloque de memoria |

Es importante, finalmente, tener en cuenta el siguiente detalle para que no tengamos confusión: no es lo mismo **(nombrePuntero + i)* que **(nombrePuntero) + i* . Ojo. En el primer caso, primero movemos el puntero un número de elementos dado por *i* (tal como hemos aprendido), y entonces, obtenemos el valor almacenado en esa nueva posición a la que acabamos de acceder. En cambio, en el segundo caso, el puntero no lo movemos de sitio: primero obtenemos el valor almacenado en la posición donde está el puntero, y a ese valor, le sumamos un número entero *i*.

Sabiendo lo explicado en los párrafos anteriores, ahora estamos en condiciones de entender el siguiente ejemplo, donde se muestra el funcionamiento de las dos funciones anteriores:

```
import "mod_say";
import "mod_mem";

Process Main()
Private
    byte pointer pbyte;
    word pointer pword;
    int  pointer pint;
    int elements = 10;
    int i;
End
Begin
//Reservo una zona de memoria de 10 bytes
    pbyte = alloc(elements);
//Reservo una zona de memoria de 20 bytes (ya que un word ocupa 2 bytes)
    pword = alloc(elements*sizeof(word));
//Reservo una zona de memoria de 40 bytes (ya que un int ocupa 4 bytes)
    pint  = alloc(elements*sizeof(int));

/*El funcionamiento de las funciones memset,memsetw y memseti se explican unos párrafos
más adelante en este mismo capítulo. Pero para entenderlo rápidamente, lo que se está
haciendo es rellenar las tres zonas de memoria reservadas todo con 0*/
    memset (pbyte,0,elements);
    memsetw(pword,0,elements); //Equivalente a memset(pword,0,elements*sizeof(word));
    memseti(pint,0,elements);  //Equivalente a memset(pint ,0,elements*sizeof(int));

/*Se escribe un valor numérico en todos los elementos de la zona de memoria reservada. Se
podría haber hecho lo mismo (ya que estamos rellenando todos los elementos con el mismo
valor) con las funciones memsetX, igual que se ha hecho más arriba. Por ejemplo, así:
memsetw(pword,345,elements); */
    for(i=0; i<elements; i++)
        pbyte[i] = 133; // pbyte[i] es lo mismo que escribir *(pbyte+i), en este bucle
        pword[i] = 345; // pword[i] es lo mismo que escribir *(pword+i), en este bucle
        *(pint+i) = 4555; //*(pint+i) es lo mismo que escribir pint[i], en este bucle
    end

    //Enseño el contenido de la zona de memoria una vez modificada
    for(i=0; i<elements; i++)
        say("byte["+i+"] = " + *(pbyte+i));
        say("word["+i+"] = " + pword[i]);
        say("int ["+i+"] = " + pint[i]);
        say("");
    end

    //Libero las diferentes zonas de memoria utilizadas
    free(pbyte);
    free(pword);
    free(pint);
End
```

| **CALLOC(INT NUMELEM, INT TAMAÑOELEM)** |
|---|
| Esta función, definida en el módulo “mod_mem”, reserva un bloque de memoria de un determinado tamaño para su posterior uso exclusivo por parte de nuestro programa, al igual que hace **alloc().** Pero a diferencia de ésta, el tamaño del bloque a reservar se especifica con dos parámetros: el número de elementos individuales que compondrán en total el bloque entero, y el tamaño de cada uno de esos elementos, en bytes.<br><br>Esta función puede devolver error (valor -1) si al intentar reservar memoria se encuentra con que no hay suficiente memoria libre disponible para el tamaño especificado. De todas maneras, este error es bastante inusual.<br><br>PARAMETROS:<br>INT NUMELEM: Número de “bloques” individuales que formarán el bloque completo a reservar |

INT TAMAÑOELEM: Número de bytes que ocupa uno de esos “bloques”individuales

VALOR RETORNADO:
POINTER: Puntero que apunta al primer elemento (el principio) del bloque de memoria reservada

El ejemplo de **calloc()** es el mismo que el de **alloc()**, pero sustituyendo una función por otra, por lo que se han obviado los comentarios repetidos:

```
import "mod_say";
import "mod_mem";

Process Main()
Private
    byte pointer pbyte;
    word pointer pword;
    int  pointer pint;
    int elements = 10;
    int i;
End
Begin
//Reservo una zona de memoria de 10 bytes
    pbyte = calloc(elements,sizeof(byte));
//Reservo una zona de memoria de 20 bytes (ya que un word ocupa 2 bytes)
    pword = calloc(elements,sizeof(word));
//Reservo una zona de memoria de 40 bytes (ya que un int ocupa 4 bytes)
    pint  = calloc(elements,sizeof(int));

    memset (pbyte,0,elements);
    memsetw(pword,0,elements);
    memseti(pint,0,elements);

    for(i=0; i<elements; i++)
        pbyte[i] = 133;
        pword[i] = 345;
        *(pint+i) = 4555;
    end

    for(i=0; i<elements; i++)
        say("byte["+i+"] = " + *(pbyte+i));
        say("word["+i+"] = " + pword[i]);
        say("int ["+i+"] = " + pint[i]);
        say("");
    end

    free(pbyte);
    free(pword);
    free(pint);
End
```

**REALLOC(POINTER PUNTERO, INT TAMAÑO)**

Esta función, definida en el módulo “mod_mem”, redimensiona un bloque de memoria previamente reservado (con **alloc()** ó **calloc()** ). En realidad, ubica un nuevo bloque de memoria, copiando el contenido del antiguo. Si el nuevo tamaño es más pequeño que el tamaño antiguo, la última parte de los datos se pierde. Si el nuevo tamaño del bloque de memoria requiere el movimiento del bloque entero (porque no cabe allí donde está), el bloque antiguo se libera automáticamente.

Esta función puede devolver error (valor -1) si al intentar reservar memoria se encuentra con que no hay suficiente memoria libre disponible para el tamaño especificado. De todas maneras, este error es bastante inusual.

PARAMETROS:
POINTER PUNTERO: Puntero que apunta al bloque de memoria ANTIGUO, el cual será redimensionado
INT TAMAÑO : Número de bytes que tendrá el NUEVO bloque de memoria a reservar

VALOR RETORNADO:
POINTER : Puntero que apunta al primer elemento (el principio) del NUEVO bloque de memoria reservada

Un ejemplo de la función **realloc()**:

```
import "mod_mem";
import "mod_say";

process main()
Private
    byte pointer pbyte;
    byte pointer pbyte2;
    int elements = 10;
    int newelements = 15;
    int i;
end
Begin
//Reservo un bloque de memoria de tamaño 10 bytes
    pbyte = alloc(elements);
//Le doy un valor numérico 13 a todos los elementos de ese bloque de memoria
    memset(pbyte,13,elements);
//Reubico el bloque de memoria anterior a otro más grande
    pbyte2 = realloc(pbyte,newelements);
/*Doy un valor numérico 16 sólo a los nuevos elementos de la parte añadida (newelements >
elements)*/
    memset(pbyte+elements,16,newelements-elements);
/*Muestro los valores de los elementos de este nuevo bloque de memoria. Los antiguos
deberán valer 13 y los nuevos 16 */
    for(i=0; i<newelements; i++)
        say("byte2["+i+"] = " + pbyte2[i]);
    end

/*Libero el bloque de memoria reservado por realloc. El bloque reservado inicialmente por
alloc,si fuera diferente,ya se liberó automáticamente cuando hubo el redimensionamiento*/
    free(pbyte2);
End
```

Otro ejemplo de **realloc()**:

```
import "mod_text";
import "mod_mem";
import "mod_key";
import "mod_proc";

type struct1
    int     var1;
    float   var2;
end

process main()
private
    struct1  st1[] =  10    ,   10.10     ,
                      5     ,   5.5       ,
                      210   ,   210.210   ,
                      3     ,   3.3       ,
                      33    ,   33.33     ,
                      12    ,   12.12     ,
                      3     ,   3.3       ;

    struct1 * st2;
    word * ivar;
    int i;
end
begin
//Reservo una zona de memoria donde pueden caber hasta tres estructuras de tipo struct1
    st2 = alloc(sizeof(struct1)*3);
```

```
/*Si el valor devuelto por alloc es igual a 0, quiere decir que no se ha logrado
encontrar un bloque para reservarlo*/
    if (st2 ==0)
        write(0,0,0,0,"Error allocando memoria, ESC para salir");
        while(!key(_ESC)) frame; end
        exit();
    end
/*Copio sólo los tres primeros elementos -no caben más- de la tabla de estructuras st1
(es decir:10,10.10 y 5,5.5 y 210,210.210)en la dirección de memoria apuntada por st2 */
    memcopy(st2, st1, sizeof(st2[0])*3);
//Los muestro
    for(i=0;i<3;i++)
        write(0,0,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
    end

    write(0,0,190,0,"Presione ESC para continuar");
    while(!key(_ESC)) frame; end
    while(key(_ESC)) frame; end
    delete_text(0);

/*Reservo una nueva zona de memoria donde pueden caber ahora hasta siete estructuras de
tipo struct1*/
    st2 = realloc(st2, sizeof(struct1)*7);
    if (st2 ==0)
        write(0,0,0,0,"Error reallocando memoria, ESC para salir");
        while(!key(_ESC)) frame; end
        exit();
    end
/*Copio ahora todos elementos definidos en la sección private de la tabla de estructuras
st1 en la dirección de memoria apuntada por st2 */
    memcopy(st2, st1, sizeof(st2[0])*7);
//Los muestro
    for(i=0;i<7;i++)
        write(0,0,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
    end

    write(0,0,190,0,"Presione ESC para continuar");
    while(!key(_ESC)) frame; end
    delete_text(0);
    free(st2);
end
```

Otras funciones de las que dispone Bennu relacionadas con la memoria (aunque no con punteros) son:

| **MEMORY_FREE()** |
| --- |
| Esta función, definida en el módulo “mod_mem”, devuelve la cantidad de memoria RAM libre existente en ese momento, en bytes. |

| **MEMORY_TOTAL()** |
| --- |
| Esta función, definida en el módulo “mod_mem”, devuelve la cantidad de memoria RAM total de la máquina, en bytes. |

Ejemplos de ambas funciones son los siguientes:

```
import "mod_mem";
import "mod_text";
import "mod_key";

process main()
private
  int memBYTE;
  int memMB;
end
```

```
begin
  write(0,10,90,3,"Memoria libre en BYTEs:");
  write(0,10,110,3,"Memoria libre en MBs: ");
  write_var(0,160,90,3,memBYTE);
  write_var(0,160,110,3,memMB);
  repeat
    memBYTE=memory_free();
    memMB=memory_free()/(1024*1024);
    frame;
  until(key(_esc))
end
```

```
import "mod_mem";
import "mod_text";
import "mod_key";

process main()
private
  int memBYTE;
  int memMB;
end
begin
  write(0,10,90,3,"Memoria total en BYTEs:");
  write(0,10,110,3,"Memoria total en MBs: ");
  memBYTE=memory_total();
  memMB=memory_total()/(1024*1024);
  write(0,160,90,3,memBYTE);
  write(0,160,110,3,memMB);
  repeat
     frame;
  until(key(_esc))
end
```

Esta funciones son muy útiles para controlar en todo momento el uso que hace nuestro juego de los recursos de la máquina (en concreto, la memoria) y poder descubrir si en algún momento de su ejecución se está cargando demasiado la máquina y agotando la memoria. Es una manera de evitar que nuestros juegos se relenticen demasiado y mantener la máquina en óptimas condiciones permanentemente. Por ejemplo, es muy interesante observar el valor que devuelve **memory_free()** justo después de haber descargado un FPG, o un FNT o un sonido con la correspondiente función *unload_**: se puede comprobar que se aumenta sensiblemente la cantidad de memoria libre, posibilitando que nuestro juego pueda funcionar con un poco más de fluidez.

Otras funciones que nos posibilitan trabajar directamente con la memoria (las cuales ya han aparecido en los códigos de ejemplo anteriores) son:

| **MAP_BUFFER(LIBRERIA,GRAFICO)** |
|---|
| Esta función avanzada, definida en el módulo "mod_mem", devuelve un puntero al buffer del gráfico que se especifica, esto es, un puntero a la zona de memoria donde se guardan los píxeles reales que forman la imagen. Cualquier manipulación del contenido de esta zona de memoria afecta directamente al contenido de la imágen.<br><br>Algunos usos interesantes de esta función, junto con **memset()** o **memsetw()** , permiten realizar efectos visuales interesantes como borrados o rellenados con bandas de color<br><br>El puntero devuelto es un puntero a BYTE para gráficos de 8 bits , un puntero a WORD para gráficos de 16 bits y un puntero a DWORD para gráficos de 32 bits.<br><br>PARAMETROS:<br>INT LIBRERIA : Número de librería FPG (los gráficos cargados con funciones de tipo *load_xxx()* o los creados con **map_new()** pertenecen a la librería 0).<br>INT GRAFICO: Número del gráfico dentro de la librería<br><br>VALOR RETORNADO: POINTER : Puntero al buffer de píxeles del gráfico |

**MEMSET (&MEM,VALOR,BYTES)**

Esta función avanzada, definida en el módulo "mod_mem", modifica gráficos de 8 bits rellenando una zona de memoria de tamaño BYTES cuyo comienzo está situado en MEM con el valor indicado por VALOR, el cual será un valor posible dentro de la paleta utilizada..

PARAMETROS:
- POINTER MEM : Puntero a una zona de memoria
- BYTE VALOR : Valor que se desea asignar a todos los bytes de esa zona
- INT BYTES : Tamaño en bytes de la zona de memoria

**MEMSETW(&MEM,VALOR,WORDS)**

Esta función avanzada, definida en el módulo "mod_mem", modifica gráficos de 16 bits rellenando una zona de memoria de tamaño WORDS cuyo comienzo está situado en MEM con el valor indicado por VALOR, el cual será un color devuelto por **rgb()** o similar.

Hay que tener presente que el tamaño total en bytes de la zona que se rellena es WORDS * 2 ya que el tamaño de cada WORD es de 2 BYTES.

PARAMETROS:
- POINTER MEM : Puntero a una zona de memoria
- WORD VALOR : Valor que se desea asignar a todos los bytes de esa zona
- INT WORDS : Tamaño en words (1 word = 2 bytes) de la zona de memoria

**MEMSETI(&MEM,VALOR,DWORDS)**

Esta función avanzada, definida en el módulo "mod_mem", modifica gráficos de 32 bits rellenando una zona de memoria de tamaño DWORDS cuyo comienzo está situado en MEM con el valor indicado por VALOR, el cual será un color devuelto por **rgb()** o similar.

Hay que tener presente que el tamaño total en bytes de la zona que se rellena es DWORDS * 4 ya que el tamaño de cada DWORD es de 4 BYTES.

PARAMETROS:
- POINTER MEM : Puntero a una zona de memoria
- INT VALOR : Valor que se desea asignar a todos los bytes de esa zona
- INT DWORDS : Tamaño en double words (1 double word = 4 bytes) de la zona de memoria

Un ejemplo de estas funciones es el siguiente. Si ejecutas el código verás que un gráfico que en principio iba a ser un simple cuadrado amarillo (generado dinámicamente con **new_map()** y pintado con **map_clear()** como hemos venido haciendo muchas veces siempre igual) estará parcialmente cubierto de color rojo: esa zona del gráfico corresponde a la zona que ocupa éste en la memoria y que ha sido rellenada de un mismo valor todos sus dwords (en este caso, el valor rgb=(255,0,0)). El tamaño de esta zona vendrá dado por los 5200 dwords especificados -que corresponde más o menos desde el principio de la imagen hasta su mitad, si ésta es de 100x100-.

```
import "mod_mem";
import "mod_text";
import "mod_key";
import "mod_map";
import "mod_video";

process main()
private
          int idpng;
/*Es "dword pointer" porque trabajamos con un gráfico de 32 bits. Con 16 bits sería "word
pointer" y utilizaríamos la función memsetw().Con 8 bits sería "byte pointer" y
utilizaríamos la función memset().*/
          dword pointer pmap;
end
begin
```

```
            set_mode(640,480,32);
             //Creo una imagen que será un cuadrado amarillo
            idpng=new_map(100,100,32);map_clear(0,idpng,rgb(255,255,0));
             //Obtengo la posición en memoria del inicio de la imagen anterior
            pmap=map_buffer(0,idpng);
//Y modifico a partir de allí el valor de 1200 dwords, poniéndolos todos a color rojo.
            memseti(pmap,rgb(255,0,0),5200);
           //Muestro el gráfico, ya modificado en memoria, por pantalla
            graph=idpng;x=320;y=240;
            while(!key(_esc))frame;end
end
```

## *Trabajar con tablas Blendop

"Blendop" proviene del apócope "Blending Operation". Es una operación de mezcla de colores (transparencias de porcentaje variable, entintado, iluminación, escala de grises, etc) que se aplica a un gráfico antes de ser dibujado. No obstante, las operaciones blendop sólo funcionan en modo de 16 bits, por ahora. Tener tablas "blendop" en modo 32 bits implicaría muchísima memoria (en concreto: 4GB x sizeof(int32) x 2 por tabla, lo cual es una locura). Y si no se tuvieran las tablas en memoria, realizar las transformaciones constantemente en tiempo de ejecución sería pesadísimo para el procesador. Sea por una razón ó por otra, las tablas "blendop" no es razonable incluirlas en modo 32 bits hasta que no haya soporte para aceleración por hardware.Si se desea una explicación más pormenorizada de la estructura interna de las tablas "blendop", se puede consultar el artículo correspondiente de la wiki de Bennu.

La idea del funcionamiento de las operaciones blend es la siguiente: primero se crea lo que se llama un tabla blendop, con la función **blendop_new()** ,la cual devuelve un identificador de dicha tabla. Dicha tabla tendrá que contener en su interior un conjunto de operaciones (de "efectos") que se definirán a continuación con las funciones **blendop_tint()** -para efectos de entintado-, **blendop_translucency()** -para transparencias-, **blendop_grayscale()** -para escala de grises-,**blendop_intensity()** -para el brillo-, etc, las cuales hacen uso del identificador de la tabla blendop previamente devuelto por **blendop_new()**. En este momento tendremos una tabla blendop que contiene una serie de efectos en cadena listos para aplicar a un/unos gráfico/s determinados. Falta decir cuáles, y podremos observar entonces el resultado. Hay diferentes maneras de asignar una tabla blendop a un gráfico: o bien con la variable local predefinida ***BLENDOP*** (si estamos hablando del gráfico de un proceso), o bien con las funciones **blendop_apply()** o **blendop_assign()** si estamos hablando de un gráfico "estándar". La diferencia entre **blendop_apply()** y **blendop_assign()** está en que mediante la primera los efectos se aplican al gráfico de forma permanente de forma que éste queda alterado incluso después de haberse descargado la tabla blendop de memoria (con **blendop_free()**), y mediante la segunda el gráfico sólo permanecerá alterado con los efectos de la tabla blendop asignada mientras ésta esté cargada en memoria.

Todas las funciones tratadas en este apartado están definidas en el módulo "mod_blendop".

### BLENDOP_NEW()

Crea una nueva tabla blendop en memoria, la cual permite crear efectos de colores y mezclas al dibujar un gráfico

Hay que tener en cuenta, no obstante, que una tabla blendop recién creada no realiza ninguna operación, por lo que usarla para dibujar no alterará el aspecto del gráfico. Es necesario especificar las características del efecto deseado empleando cualquiera de las funciones blend_XXX(). La mayoría de estas funciones son acumulables sobre una misma tabla, por lo que es posible especificar un efecto de coloreado con **blendop_tint()** y seguidamente un porcentaje de transparencia con **blendop_translucenty()** , por ejemplo.

Una tabla blendop puede usarse de tres maneras diferentes, como ya hemos dicho:

- Activarla en un proceso, asignando el identificador devuelto por esta función a la variable local ***BLENDOP***. Esto permite dibujar cualquier proceso con el efecto de color deseado.
- Asignarla a un gráfico, empleando la función **blendop_assign()**. Cualquier uso posterior de ese gráfico empleará la tabla blendop para dibujarlo, (aunque la variable local ***BLENDOP*** tiene preferencia si está presente -y los efectos no se acumularán-.), siempre y cuando la tabla continúe cargada en memoria.

- Aplicarla a un gráfico, empleando la función **blendop_apply()**. Los puntos del gráfico serán modificados de acuerdo con la tabla blendop de forma permanente.

Una tabla blendop ocupa una cantidad considerable de memoria (256K) por lo que se aconseja no excederse en su uso. Por ejemplo, un programa que cree 20 tablas blendop necesitará unos 5M de RAM sólo para almacenarlas. Si una tabla sólo se va a usar temporalmente, se recomienda emplear **blendop_free()** para liberarla.

VALOR DE RETORNO: INT : Identificador de la nueva tabla BLENDOP

**BLENDOP_TINT ( BLENDOP, CANTIDAD, R, G, B )**

Aplica un color a una tabla blendop. Dada una tabla blendop creada con **blendop_new()** , esta función realiza una mezcla entre el color cuyas componentes recibe como parámetros, y los colores del gráfico (todo ello en el gráfico origen de la operación). El valor de cantidad especificado como segundo parámetro indica la cantidad del nuevo color que se fusionará en cada pixel. El resultado será que cada color del gráfico se transforma según la fórmula (CANTIDAD * NUEVO + (1.0 - CANTIDAD) * VIEJO).

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones blendop

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP
FLOAT CANTIDAD: Porcentaje relativo al nuevo color (0.0 a 1.0)
INT R : Componente Roja del color (de 0 a 255)
INT G : Componente Verde del color (de 0 a 255)
INT B : Componenete Azul del color (de 0 a 255)

**BLENDOP_INTENSITY ( BLENDOP, INTENSIDAD )**

Aplica brillo a una tabla blendop. Dada una tabla blendop creada con **blendop_new()**, esta función aplica un factor de intensidad a todos los colores del gráfico de origen, es decir, multiplica todas sus componentes por el valor especificado como segundo parámetro. Un valor de 0.5 oscurecería todos los colores a un 50%, mientras un valor de 2.0 aumentaría la intensidad de los colores al doble aproximadamente.

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones blendop

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP
FLOAT INTENSIDAD: Cantidad de brillo

**BLENDOP_TRANSLUCENCY ( BLENDOP, OPACIDAD )**

Aplica transparencia a una tabla blendop Dada una tabla blendop creada con **blendop_new()** , esta función aplica un valor de transparencia según el parámetro de opacidad. 1.0 equivale a un dibujo completamente opaco (tal como es creada la tabla), mientras 0.0 dibujaría un gráfico completamente transparente. Este efecto es semejante al proporcionado por la variable local ***ALPHA*** , aunque mucho más preciso.

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones blendop.

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

FLOAT OPACIDAD: Cantidad de opacidad, entre 0.0 y 1.0

**BLENDOP_GRAYSCALE ( BLENDOP, METODO )**

Aplica conversión a escala de grises a una tabla blendop. Dada una tabla blendop creada con **blendop_new()**, esta función aplica una fórmula al gráfico original que convierte todos los colores en su equivalente en escala de grises.

Para realizar esta operación se soportan tres métodos de conversión diferentes:

1: Luminancia. El color resultante tiene una intensidad de 0.3*R + 0.59*G + 0.11*B. Normalmente, éste es el método que ofrece una mayor nitidez en imágenes fotográficas.
2: Desaturación. El color resultante es la media de las tres componentes R, G y B. Recomendable para gráficos oscuros o coloreados a mano con cierta variedad de color.
3: Máximo. El color resultante es el máximo entre las tres componentes. Recomendable para gráficos de líneas con pocos colores de diferencias marcadas.

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP
INT METODO: Método a aplicar (1,2 ó 3)

**BLENDOP_IDENTITY ( BLENDOP )**

Reinicia una tabla blendop. Dada una tabla blendop creada con **blendop_new()**, esta función la reinicia, es decir, la devuelve a sus valores originales (consistentes en no modificar el gráfico al dibujarlo).

En ocasiones puede ser útil cambiar las características de una tabla blendop, sin emplear los efectos ya acumulados. Esta función la devuelve a su estado inicial, tal como fue creada.

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

**BLENDOP_SWAP ( BLENDOP )**

Intercambia el efecto de una operación en una tabla blendop Dada una tabla blendop creada con **blendop_new()**, esta función intercambia las operaciones realizadas con el gráfico de origen y las realizadas con el de destino. Las funciones **blendop_tint()**, **blendop_grayscale()** y **blendop_intensity()** modifican sólo el gráfico original. Utilizando esta función, es posible hacer que las operaciones acumuladas alteren el gráfico destino.

Dicho en palabras menos técnicas: esta función altera el orden de la tabla blendop de manera que los efectos se apliquen sobre el fondo en lugar de sobre el gráfico, permitiendo efectos de tipo "lente".

Es preciso entender que todos los pixels a 0 en un gráfico son ignorados por las rutinas de dibujo: una tabla blendop que haga modificaciones sobre el gráfico de destino sólo afectará a los pixels en pantalla que haya debajo de pixels que no estén a cero en el gráfico a dibujar.

Una tabla blendop recién creada tiene una opacidad del 0% para el gráfico de destino. Si se intercambia con esta función sin haber llamado a **blendop_translucency()** , el efecto resultante será que los pixels del gráfico a dibujar no serán visibles de por sí, pero los pixels que haya bajo todos ellos serán modificados según las operaciones que hubiese en la tabla blendop acumuladas antes de llamar a **blendop_swap()** Si se desea un gráfico parcialmente visible, que haga algún tipo de efecto (como oscurecer o entintar) sobre la zona de pantalla que haya bajo él, es preciso llamar a **blendop_translucency()** antes de **blendop_swap()**.

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones blendop

PARÁMETROS: INT BLENDOP : Identificador de una tabla BLENDOP

**BLENDOP_FREE ( BLENDOP )**

Destruye una tabla blendop Esta función libera la memoria ocupada por una tabla blendop. Es responsabilidad del programador asegurarse de que la tabla blendop no se encuentre asignada a ningún gráfico ni a ninguna variable ***BLENDOP*** de ningún proceso en el momento de destruirla. Lo contrario se considera una condición de error grave y puede dar resultados impredecibles.

PARÁMETROS: INT BLENDOP : Identificador de una tabla BLENDOP

**BLENDOP_ASSIGN ( LIBRERÍA, GRÁFICO, BLENDOP )**

Asigna una tabla blendop a un gráfico Dada una tabla blendop creada con **blendop_new()**, esta función la asigna a un gráfico. Cualquier dibujo posterior del gráfico en pantalla, que no sea en un proceso que contenga su propia tabla blendop en la variable local ***BLENDOP*** , utilizará la tabla blendop para dibujarlo.

Si se especifica 0 como identificador blendop, cualquier tabla blendop asignada al gráfico dejará de usarse.

No es válido liberar una tabla blendop de memoria (con ***blendop_free()*** ) cuando aún existan gráficos que la tengan asignada.

PARÁMETROS:

INT LIBRERIA: Número de librería FPG
INT GRAFICO : Número de gráfico dentro de la librería
INT BLENDOP : Identificador de una tabla BLENDOP

**BLENDOP_APPLY ( LIBRERÍA, GRÁFICO, BLENDOP )**

Aplica una tabla blendop a los pixels de un gráfico Esta función modifica los pixels de un gráfico, aplicando las operaciones que la tabla de blendop indica que deben hacerse sobre el gráfico original. La transparencia se aplicará como oscurecimiento (mezclando el mapa con el negro).

El gráfico quedará modificado en memoria sin necesidad de seguir manteniendo la tabla blendop en memoria y sin hacerle ninguna referencia a la misma.

PARÁMETROS:

INT LIBRERIA: Número de librería FPG
INT GRAFICO : Número de gráfico dentro de la librería
INT BLENDOP : Identificador de una tabla BLENDOP

Este tema es mucho más sencillo de lo que parece. Empecemos con un ejemplo muy sencillo, el cual se aplica a un gráfico una tabla "blendop" de entintado en rojo cada vez que se pulsa la tecla Space (y se deja de aplicar cuando ésta se suelta). Este mismo ejemplo levemente modificado se podría utilizar para por ejemplo oscurecer un gráfico (aplicándole a la tabla "blendop" de entintado por ejemplo los valores 0.5, 0, 0 y 0 a los cuatro últimos parámetros de la función **blendop_tint()**: pruébalo) cuando ocurra alguna circunstancia determinada en nuestro juego.

```
import "mod_blendop";
import "mod_map";
import "mod_video";
import "mod_key";

Process main()
Private
```

```
    int blendTable;
End
Begin
    set_mode(320,240,16);
    x = 160; y = 120;
    graph = new_map(100,100,16); map_clear(0,graph,RGB(255,255,255));

    blendTable = blendop_new();
    blendop_tint(blendTable,1,255,0,0);
    Repeat
        if (key(_space))
            blendop_assign(0,graph,blendTable);
        else
            blendop_assign(0,graph,NULL);
        end
        frame;
    Until(key(_ESC))
    unload_map(0,graph);
End
```

Con el siguiente ejemplo se verá más detalladamente cómo funciona todo. Consiste simplemente en mostrar un gráfico -generado a partir de **write_in_map()**- asociado al proceso "bola()" de cinco maneras distintas. La manera que se muestra en el centro es el gráfico sin habérsele aplicado ningún tipo de tabla blendop. La de la esquina superior izquierda es el producto de haber aplicado al gráfico inicial un efecto de tinte, el cual se puede variar cambiando los valores de las variables *c,r,g* y *b*. La de la esquina superior derecha es el producto de haber aplicado al gráfico inicial un efecto de transparencia, el cual se puede variar cambiando los valores de la variable *c*. La de la esquina inferior izquierda es el producto de haber aplicado al gráfico inicial un efecto de escalado de grises, el cual se puede variar cambiando los valores de la variable *m*. La de la esquina inferior derecha es el producto de haber aplicado al gráfico inicial el efecto de brillo, el cual se puede variar cambiando los valores de la variable *c*.

En este ejemplo no se pone en práctica, pero evidentemente, a una misma tabla blendop se puede aplicar más de un efecto (tinte, transparencia,etc) mediante las funciones correspondientes, para lograr un resultado sumativo cuando se le asigne a un gráfico.

```
import "mod_blendop";
import "mod_video";
import "mod_text";
import "mod_map";
import "mod_proc";
import "mod_key";

Global
//CAMBIAR ESTOS VALORES PARA OBSERVAR OTROS EFECTOS
  Float c=0.5; //Cantidad de tintado, de transparencia y de brillo (de 0.0 a 1.0)
  byte m=1; //Método de escalado de grises (1,2 ó 3)
  byte r=0; //Cantidad de rojo para el tinte
  byte g=255; //Cantidad de verde para el tinte
  byte b=0; //Cantidad de azul para el tinte
end

process main()
Private
  int ble;
End
Begin
  set_mode(640,480,16);
  //Ponemos un fondo de color
  fondo();

  //Antes de aplicar ninguna tabla blendop
  bola(320,240,0);

  //Aplicamos un tinte determinado según el valor de c,r,g y b
  write(0,60,20,4,"Tinte");
  ble=blendop_new();
  blendop_tint(ble,c,r,g,b);
```

```
  //blendop_swap(ble);
  bola(60,50,ble);

 //Aplicamos una transparencia determinada según el valor de c
  write(0,550,20,4,"Transparencia");
  ble=blendop_new();
  blendop_translucency(ble,c);
  bola(550,50,ble);

//Aplicamos un escalado de grises según el método determinado por m
  write(0,60,430,4,"Escala de grises");
  ble=blendop_new();
  blendop_grayscale(ble,m);
  bola(60,450,ble);

//Aplicamos un brillo determinado según el valor de c
  write(0,550,430,4,"Brillo");
  ble=blendop_new();
  blendop_intensity(ble,c);
  bola(550,450,ble);

  Repeat
    Frame;
  Until(key(_esc))
  let_me_alone();
  blendop_free(ble);
End

Process bola(x,y, int ble)
Begin
  graph = write_in_map(0,"ABCD",4);
  blendop_assign(0,graph,ble);
  /*Otra manera de hacer lo mismo que la línea anterior es  utilizar la variable local
BLENDOP, así: blendop=ble;
  La otra alternativa, blendop_apply no sé por qué hace petar  el programa.*/
  Loop
    Frame;
  End
End

Process fondo()
begin
graph=new_map(640,480,16);
map_clear(0,graph,rgb(200,30,100));
x=320;
y=240;
z=1;//Para estar por debajo del proceso bola
loop
          frame;
end
end
```

Otro ejemplo básico de funcionamiento de las tablas blendop puede ser el siguiente.

```
import "mod_blendop";
import "mod_video";
import "mod_mouse";
import "mod_proc";
import "mod_map";
import "mod_key";
import "mod_draw";
import "mod_text";
import "mod_screen";

Process Main()
Private
    int bo_tintred;
    int bo_tintredAndTranslucent;
```

```
    int bo_intense;
    int bo_destintense;
    int bo_grayscale;
    int bo_destgrayscale;
    int map;
End
Begin

    set_mode(320,200,16);

    //Inicializo unas cuantas tablas blendop
    bo_tintred = blendop_new();
    bo_tintredAndTranslucent = blendop_new();
    bo_intense = blendop_new();
    bo_grayscale = blendop_new();
    bo_destintense = blendop_new();
    bo_destgrayscale = blendop_new();

    //Tinto de rojo
    blendop_tint(bo_tintred,0.5,100,0,0);

    //Tinto de rojo con transparencia
    blendop_tint(bo_tintredAndTranslucent,0.5,250,0,0);
    blendop_translucency(bo_tintredAndTranslucent,0.5);

    //Modifico la intensidad
    blendop_intensity(bo_intense,1.5);

    //Muestro la intensidad de destino
    blendop_intensity(bo_destintense,1.5);
    blendop_swap(bo_destintense);

    //Transformo a escala de grises
    blendop_grayscale(bo_grayscale,1);

    //Muestro la escala de grises del destino
    blendop_grayscale(bo_destgrayscale,3);
    blendop_swap(bo_destgrayscale);

    //Creo una nueva imagen
    map = some_map(80,80,16);

    //Pinto el fondo. Utilizo la palabra clave "background" para referirme al gráfico 0.
    map_clear(0,background,rgb(50,150,150));

    //Muestro el efecto de las diferentes operaciones blend
    a( 50, 50,map,0,"original");
    a(150, 50,map,bo_tintred,"red");
    a(250, 50,map,bo_tintredAndTranslucent,"red+transparent");
    a( 50,150,map,bo_intense,"intense");
    a(150,150,map,bo_destintense,"dest instense");
    a(250,150,map,bo_grayscale,"grayscale");

    //Se le da al proceso actual una operación blend también
    blendop_assign(0,map,bo_destgrayscale);
    graph = map;

    Repeat
        x = mouse.x;
        y = mouse.y;
        if(mouse.left)
            z = -1;
        else
            z = 1;
        end
        frame;
    Until(key(_ESC))
```

```
OnExit
    let_me_alone();
End


/*Este proceso clonará su propio gráfico y se le asignará al gráfico clonado la tabla
blendop especificada por parámetro. También se pondrá un texto de fondo, pasado también
por parámetro.*/

Process a(x,y,graph, int blendtable, string text)
Private
   int fondo;
End
Begin
    z = 0;
    //Clono el gráfico
    graph = map_clone(0,graph);

    //Y le asigno la table blend pasada por parámetro
    blendop_assign(0,graph,blendtable);

    //Pinto un texto en el fondo
    set_text_color(rgb(100,200,250));
    fondo=write_in_map(0,text,1);
    put(0,fondo,x,y+1+map_info(0,graph,G_HEIGHT)/2);
    unload_map(0,fondo);

    Loop frame; End
End

/*La siguiente función crea un gráfico con cuatro esquinas coloreadas de forma diferente.
Los parámetros son la anchura, altura y profundidad del gráfico a crear. Devuelve el
identificador del gráfico creado.*/

Function int some_map(int ancho, int alto, int prof)
Private
    int grafico;
End
Begin
    grafico = new_map(ancho,alto,prof);
    map_clear(0,grafico,rgb(50,100,150));
    drawing_map(0,grafico);
    drawing_color(rgb(200,50,100));
    draw_box(0,0,x/2,y/2);
    drawing_color(rgb(0,250,100));
    draw_box(x/2,0,x,y/2);
    drawing_color(rgb(150,50,250));
    draw_box(x/2,y/2,x,y);
    return grafico;
End
```

Un uso de las tablas blendop está en el ámbito de generación de sombras. Si se dispone de un gráfico ya visible, a partir de él se puede generar una sombra suya sin tener que disponer de otro gráfico diferente: con las tablas blendop se puede lograr un efecto suficiente. Por ejemplo, en el código siguiente tenemos un proceso -cuyo gráfico se genera dinámicamente: es un cuadrado dentro de otro, el cual él mismo genera su propia sombra invocando al proceso "sombra()". Este proceso lo único que hace es copiar una serie de valores de variables locales del padre (mismo graph, mismo angle, mismo size, misma xetc) y alterar levemente otra serie (flags para ver la sombra bocaabajo, z para ver la sombra por debajo del gráfico original,y para verse en la parte inferior...). Lo realmente interesante es que además, al gráfico perteneciente a este proceso se le aplica una tabla blendop que contiene un conjunto de efectos en cadena como son un escalado de grises, un nivel de transparencia medio, un brillo importante y un entintado ténue. El resultado puede variarse a nuestro gusto dependiendo del valor de los parámetros que pongamos a las diferentes funciones blendop.

```
import "mod_map";
import "mod_video";
import "mod_blendop";
```

```
import "mod_key";
import "mod_proc";

process main()
begin
set_mode(640,480,16);
proceso();
loop
          frame;
end
end

process proceso()
private
  int grafico;
end
begin
//Dibujo el gráfico del proceso
  graph=new_map(100,100,16);map_clear(0,graph,rgb(140,26,48));
  grafico=new_map(50,50,16);map_clear(0,grafico,rgb(40,126,148));
  map_put(0,graph,grafico,50,20);
  size=50;
  x=320; y=100;

  sombra();
  loop
       if(key(_right)) x=x+10; end
          if(key(_left)) x=x-10; end
          frame;
  end
end

//Pensado para ser llamado desde el proceso del que va a ser sombra
process sombra()
private
  int sombra;
end
begin
sombra=blendop_new();
blendop_grayscale(sombra,1);
blendop_translucency(sombra,0.5);
blendop_intensity(sombra,0.9);
blendop_tint(sombra, 0.7, 100, 100, 100);
blendop=sombra;

graph=father.graph;
angle=father.angle;
size_y=75; //Lo ponemos un poco mas "achaparrao" que el proceso original
//size=father.size;
flags=father.flags+2; //para que este boca abajo
z=father.z+1;
y=father.y+150; //si el grafico tiene su centro virtual en la base no haría falta sumar
nada
repeat
  x=father.x;
  frame;
until(key(_esc))
exit();
end
```

El código anterior fácilmente se podría haber modificado para obtener un proceso “sombra()” -con los parámetros que se consideraran necesarios, como la distancia del proceso padre, el tipo de tabla “blendop” a aplicar, el tamaño de la sombra, etc- que permitiera generar de forma cómoda y fácil sombras de diferentes procesos, reutilizando así casi todo el código.

El siguiente ejemplo incluye la utilización de tablas blendop para generar sombras, pero además se demuestra el uso de regiones. La idea es que el usuario puede manejar un personaje a derecha e izquierda. Este personaje estará

oculto en la noche -es decir, se verá negro-, siempre que no entre en una región definida que representa un muro o pared. En ese momento, el personaje se verá tal cual es -mostrando todos sus colores-, y ahora la sombra se verá reflejada en la "pared" a un tamaño mayor, como si al personaje le estuviera iluminando un gran foco.

```
import "mod_blendop";
import "mod_draw";
import "mod_key";
import "mod_screen";
import "mod_video";
import "mod_map";
import "mod_proc";

global
      int bl;
end

process main()
begin
set_mode(640,480,16);
set_fps(50,0);
define_region(1,380,0,260,395);  //La "pared"

drawing_map(0,0);
//Pinto un rectángulo de un color para el fondo de la pantalla (representa la "noche")
drawing_color(rgb(100,150,100));
draw_box(0,0,640,480);
//Pinto un rectángulo de otro color del tamaño de la región 1 para poder visualizarla.
/*El orden con el que se escriben en el código las primitivas es importante: la última
primitiva se pintará por encima de las anteriores.*/
drawing_color(rgb(100,0,100));
draw_box(380,0,640,395);

/*Esta tabla blendop será usada para tintar de negro las dos sombras que generará nuestro
personaje*/
bl=blendop_new();
blendop_tint(bl,1,4,4,4);

personaje();
end

process personaje()
begin
graph=dibujopersonaje();
x=100;y=350;
sombra1();
sombra2();
repeat
          switch (scan_code)
              case _left: x=x-10; end
              case _right:x=x+10; end
          end
          frame;
until(key(_esc))
 /*Si la tabla blendop se utilizara sólo en un proceso, se podría poner en éste una
cláusula ONEXIT con la liberación de la tabla*/
blendop_free(bl);
exit();
end

process sombra1()
begin
//Para que al moverse el padre no vaya por delante
priority=father.priority -1;
graph=father.graph;
/*Aplicamos la tabla blendop a la sombra, con lo que su gráfico aparecerá todo de color
casi negro*/
blendop=bl;
//La sombra aparecerá encima del personaje,para que cuando sombra1 esté visible,lo tape
```

```
z=father.z-1;
loop
//Aquí es donde se aplica de forma práctica el priority.
          x=father.x;
          y=father.y;
//Si está dentro de la región 1, la sombra1 desaparece. Si está fuera, vuelve a aparecer
          if(out_region(id,1)==0) alpha=0; else  alpha=255; end
          frame;
end
end

process sombra2()
begin
size=200;
z=20;
region=1;
blendop=bl;
graph=father.graph;
loop
          x=father.x;
          y=father.y-50;
//La transparecia de la sombra debería disminuir a medida que X crece.
          alpha=x-380;
          frame;
end
end

//Obtengo el dibujo del personaje (primitivas gráficas: círculos y líneas)
function dibujopersonaje()
private
     int midibujo;
end
begin
midibujo=new_map(100,100,16);
drawing_map(0,midibujo);
drawing_color(rgb(255,0,0));
draw_line(0,0,100,100);
drawing_color(rgb(0,255,0));
draw_line(0,100,100,0);
drawing_color(rgb(0,0,255));
draw_line(50,0,50,100);
drawing_color(rgb(255,255,0));
draw_line(0,50,100,50);
drawing_color(rgb(0,0,255));
draw_fcircle(50,50,30);
drawing_color(rgb(0,255,0));
draw_fcircle(50,50,20);
drawing_color(rgb(255,0,0));
draw_fcircle(50,50,10);
return midibujo;
end
```

Por último, vamos a ver ejemplos de tres funciones blendop, cuyo uso sea tal vez el más difíciles de comprender: **blendop_identity()**, **blendop_swap()** y **blendop_free()**

Prueba este ejemplo:

```
import "mod_blendop";
import "mod_key";
import "mod_effects";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_rand";
```

```
process main()
begin
          set_mode(640,480,16);
          fill_map();
          loop if(key(_esc)) exit(); end frame; end
end

Process fill_map()
private
           int idpng;
           int i;
           int ble;
end
begin
idpng=new_map(50,50,16);map_clear(0,idpng,rgb(255,0,0));
blur(0,idpng,0); //Un efecto extra
ble=blendop_new();
graph=idpng;x=320;y=240;
loop
  blendop_identity(ble);
  blendop_tint(ble,0.1,rand(0,255),rand(0,255),rand(0,255));
  blendop_assign(0,idpng,ble);
  frame;
end
end
```

Verás que lo que ocurre es que la imagen del proceso va cambiando su tinte de forma consecutiva e infinita. ¿Por qué?. En el loop del proceso "fill_map()", una vez ya creada antes la tabla blendop, lo primero que hago es aplicar un identity. En la primera iteración no tiene sentido "limpiar" la tabla blendop porque de entrada ya se crea "limpia", pero ahora veremos que en sucesivas iteraciones ésta es necesaria. Seguidamente, aplico un tinte a la tabla blendop, y asigno ésta al gráfico en memoria que de hecho corresponde al gráfico del proceso. Justo después viene el fotograma, con lo que se verá el gráfico tinteado. A la siguiente iteración es cuando la función identity toma todo su sentido, porque limpia el tinte previo de la tabla blendop y la deja tal como si estuviera acabada de crear. Una vez hecho esto, vuelvo a aplicar otro tinte sobre la tabla "virgen", lo vuelvo a aplicar al gráfico y lo vuelvo a mostrar.

¿Qué pasaría si no estuviera la línea con la función identity? Que a cada iteración se iría añadiendo un tinte sobre otro en la tabla blendop. Y el efecto resultante sería que en unos pocos frames, el gráfico se vería de un color marronizo homogéneo, ya que a cada tinte que se aplica se va homogeneizando los distintos colores de la imagen: si se aplican muchos tintes seguidos se llega a la homogeneización total. Pruébalo.

Otro ejemplo diferente para mostrar la utilidad de esta función es el siguiente: en él se mostrará el mismo gráfico sobre el que se ha asignado una tabla blendop formada por un tinte de color verde. Cuando el usuari pulse ENTER, se mostrará el gráfico sin tinte (de color rojo) ya que se "limpiará" la tabla. Pasados tres segundos, se volverá a tintear la tabla blendop y el gráfico se volverá a ver afectado por ese cambio de tinte. Fíjate en el detalle de haber usado **map_clone()** para poder aplicar la tabla blendop a un gráfico que no es el gráfico originalmente cargado desde el archivo. Esto se suele hacer para preservar dicho gráfico de posibles modificaciones y mantenerlo siempre tal cual, haciendo que cualquier modificación recaiga sobre clones suyos.

```
import "mod_blendop";
import "mod_key";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_timers";
import "mod_rand";

process main()
private
  int bldop;
  int idpng;
  int bid;
end
begin
  set_mode(320,200,16);
```

```
  idpng=new_map(50,50,16);map_clear(0,idpng,rgb(255,0,0));
  write(0,160,70,4,"ENTER = IDENTITY Tabla BLENDOP");
  graph=map_clone(0,idpng);x=160;y=100;
  bldop=blendop_new();
  repeat
    if(key(_enter) && bid==0)
      blendop_identity(bldop);
      blendop_assign(0,graph,bldop);
      bid=1;
      timer=0;
    end
    if(timer>=300 && bid==1)
      blendop_tint(bldop,0.50,0,255,0);
      blendop_assign(0,graph,bldop);
      bid=0;
    end
    frame;
  until(key(_esc))
end
```

Para ver el efecto que causa la función **blendop_swap()** sobre un gráfico, veremos el siguiente código., en el cual habrá dos gráficos involucrados: uno correspondiente a un proceso y otro al fondo de pantalla, el cual sería conveniente que variado variado (es decir, que no sea un color homogéneo, que tenga elementos distintivos) para poder apreciar el efecto mejor. He aquí el ejemplo: pulsando enter aplicaremos y desaplicaremos alternativamente el efecto swap a la tabla blendop, y en consecuencia, a su gráfico asociado.

```
import "mod_blendop";
import "mod_key";
import "mod_video";
import "mod_map";
import "mod_timers";
import "mod_screen";

process main()
private
  int bldop;
  int idpng;
  int bswap;
  int fondo;
end
begin
  set_mode(320,240,16);
  fondo=new_map(320,240,16);map_clear(0,fondo,rgb(255,255,0));
  put_screen(0,fondo);

  idpng=new_map(40,40,16);map_clear(0,idpng,rgb(100,100,100));
  graph=map_clone(0,idpng); x=160; y=100;
  bldop=blendop_new();
  blendop_tint(bldop,0.55,0,255,0);

  repeat
    if(key(_enter) && bswap==0)
      while(key(_enter)) frame; end
      blendop_swap(bldop);
      blendop_assign(0,graph,bldop);
      bswap=1;
    end
    if(key(_enter) && bswap==1)
      while(key(_enter)) frame; end
      blendop_swap(bldop);
      blendop_assign(0,graph,bldop);
      bswap=0;
    end
    frame;
  until(key(_esc))
end
```

El siguiente código es un ejemplo de uso de la función **blendop_free()**. Si lo ejecutas, aparentemente funciona muy similar al ejemplo que hemos visto antes con la función **blendop_identity()**, pero no te engañes: en aquel ejemplo no destruíamos la tabla blendop, simplemente la limpiábamos cada vez; en cambio, en este ejemplo, cada vez que pulsamos ENTER estamos destruyendo completamente de la memoria o generando la tabla de nuevo completamente desde cero.

```
import "mod_blendop";
import "mod_key";
import "mod_video";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_timers";
import "mod_rand";

process main()
private
  int bldop;
  int idpng;
  int bfree;
end
begin
  set_mode(320,200,16);
  idpng=new_map(50,50,16);map_clear(0,idpng,rgb(255,0,0));
  write(0,160,70,4,"ENTER = Borra/Crea la Tabla BLENDOP");
  graph=map_clone(0,idpng);x=160;y=100;
  bldop=blendop_new();
  repeat
    if(key(_enter) && bfree==0)
      while(key(_enter)) frame;end
/*Línea importante: retiro la asignación que tenía el gráfico con la tabla blendop, antes
de destruir ésta.Si no hago esto, el blendop_free dará error porque para destruir una
tabla blendop antes se han de desasociar
de cualquier gráfico o proceso ligado a ella.*/
      blendop_assign(0,graph,0);
      blendop_free(bldop);
      bfree=1;
    end
    if(key(_enter) && bfree==1)
       while(key(_enter)) frame;end
/*La tabla bldop ahora mismo no existe. La tengo que volver a crear,configurar el tinte y
asignárselo al gráfico, como al principio del programa*/
       bldop=blendop_new();
       blendop_tint(bldop,0.55,0,255,0);
       blendop_assign(0,graph,bldop);
       bfree=0;
    end

    frame;
  until(key(_esc))
end
```

## *Trabajar con tiles

En el capítulo-tutorial donde se explicó cómo programar un juego RPG, ya vimos que no era buena idea cargar inmensos gráficos scrolleables representando a todo nuestro mundo imaginario, (ya que consumen vertiginosamente los recursos de la máquina y hacen que nuestro juego se ralentice sensiblemente) sino que era más conveniente utilizar los llamados "tiles".

Recuerda que un escenario “tileado” es un escenario donde se usan muchas piezas pequeñas que encajan entre sí en vez de una única imagen grande. Por ejemplo, en vez de hacer un gráfico enorme de campo verde con dos árboles se prepara un cuadrado verde y un cuadrado con un árbol y con eso se monta la imagen en tiempo de ejecución.

De esta forma ahorras mucha memoria, ya que necesitas tener menos tamaño total de gráficos cargados, y haces el juego más ligero en lo que a uso de CPU se refiere, ya que no hay nada fuera de pantalla (se pinta sólo lo que queda dentro de la pantalla).Además, si el decorado a "tilear" es un poco repetitivo, se puede ahorrar muchísima memoria, ya que si te fijas verás que los decorados son una repetición contínua de unos pocos tipos diferentes de baldosas: es mucho más económico dibujar 20 baldosas distintas y decirle al programa dónde colocarlas que guardar enormes imágenes de cada fase, que ocuparían mucho. A cambio es un poco más complicado de programar, ya que hay que controlar todas esas pequeñas piezas en vez de una grande, pero eso pasa siempre que se optimiza algo.

En aquél capítulo-tutorial simplemente se introdujo el concepto de "tile" tal como ha sido expresado en el párrafo anterior, pero no se realizó ningún ejercicio práctico de creación y manejo de escenarios tileados. Es lo que vamos a hacer en este apartado. A continuación se mostrará el código que implementa un escenario tileado muy básico, por donde se moverá una nave espacial (pero sin scroll). Para poderlo visualizar, lo que debes hacer primero es crear los tiles y meterlos en un FPG, llamado "tiles.fpg". En el ejemplo hemos creado 11 tiles de 64x64 píxeles (para que quepan 10 tiles en horizontal y 7.5 en vertical, ya que usaremos una resolución de 640x480) , con códigos del 001 al 011, los cuales pueden representar partes de un escenario cualquiera: un campo sembrado, un bosque, una montaña,etc. El gráfico de la nave lo generaremos dinámicamente en el código.

```
import "mod_text";
import "mod_key";
import "mod_map";
import "mod_scroll";
import "mod_video";
import "mod_proc";

Global
    Int Fpg_System;
    Int mapa_transp;

/*He asignado a cada elemento de la tabla el código de un tile concreto dentro del FPG.
La tabla representa el mapa en sí, y los valores que se le dan al principio es para crear
el mapa inicial del juego. El mapa estará formado por tiles colocados en distintas
posiciones, y en cada una de ellas aparecerá el tile que tenga el mismo código dentro del
FPG que el que aparece en el elemento correspondiente de esta tabla.También se podría
haber hecho con una tabla de dos dimensiones (con filas y columnas).
La tabla la usaremos en el código de tal manera que los tiles aparecerán en la parte
superior de la ventana son los que ocupan las posiciones [0-19] de la tabla, los tiles de
la fila inferior son los que ocupan las posiciones [20-39], y así.En el mapa inicial, los
tiles que se visualizan en la fila más inferior (que aparecen partidos por la mitad) son
los tiles correspondientes a los elementos [120-139]. Como queremos que el mapa sea más
grande que lo que se logra ver en pantalla al principio, he hecho que la tabla tenga más
tiles.Esto es así para lograr un efecto de scroll que se note cuando la nave se mueva a
algún extremo de la pantalla, ya que los tiles se reubicarán.
Los últimos elementos de la tabla son los del rango [220-239]. */

    Byte  Tiles[239]=
    1,1,2,10,11,4,1,1,1,1,2,3,9,3,8,3,3,3,4,2,
    3,3,3,10,11,3,3,3,3,4,2,3,9,3,8,3,3,3,4,2,
    3,3,3,10,11,3,3,3,3,4,2,3,9,3,8,5,6,6,7,9,
    1,1,2,10,11,3,3,3,3,3,3,3,9,3,8,5,6,6,7,9,
    1,1,2,10,11,5,6,6,6,7,9,3,9,3,8,5,6,6,7,9,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,9,3,9,3,9,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3;
End
Process Main()
Private
End
Begin
    set_mode(640,480);
    fpg_system=load_fpg("tiles.fpg");
    Mapeado_tile();
    Nave();
```

```
    write(0,160,40,0,"Use los cursores para mover la nave");
    write(0,160,60,0,"Esc para Salir");
    Loop
        Frame;
        If(key(_esc)) exit();End
    End
End


//Éste es el proceso que se encarga de crear el mapa tileado
Process Mapeado_tile()
Private
/*Estas dos variables las defino para referirme a las columnas y filas de la tabla que
tenemos definida en global*/
    int Fila;
    int Columna;
/*Region_act y ultima_region son las que controlan cuando deben actualizarse los tiles en
el scroll.NO ESTA IMPLEMENTADO*/
    Int region_act;
    Int ultima_region;
End
Begin
/*Inicialmente ponemos los tiles de la pantalla. Aunque tal como se ha comentado, sólo
caben 7.5 tiles verticalmente y 10 tiles horizontalmente, la tabla de tiles tiene 20x12
elementos para que sea un poco mayor que la pantalla y se pueda pintar algún tiles de más
por fuera, por si acaso*/
    From fila= 0 To 11;
                From columna= 0 To 19;
/*Fíjate la manera que usamos (muy típica cuando se trabaja con tiles) de acceder a un
elemento determinado de la tabla. Las filas están multiplicadas por 20 porque 20 es el
número de elementos de cada fila; de esta manera, primero nos situamos en una fila
determinada, y una vez allí, nos movemos a la columna que queremos. Es pues un movimiento
"en 2 pasos": nos saltamos de golpe primero todas las filas que no nos interesan para
llegar a la nuestra, y una vez allí, nos movemos a la columna que queremos. Por ejemplo,
si queremos saber qué gráfico ha de haber en la quinta fila (número 4) octava columna
(columna 7), el elemento de la tabla correspondiente será 4*20+7 = 87.
Por otro lado, las coordenadas escritas en los dos últimos parámetros de map_put
-correspondientes al centro del tile- se multiplican por el tamaño de la baldosa, para
cuadrar el dibujo. La primera baldosa de la esquina superior izquierda tendrá como punto
central el (0,0); la segunda el (64,64), la tercera el (128,128) y así.*/
                    map_put(0,0,tiles[(fila*20)+columna],columna*64,fila*64);
                End
    End

    Loop
        //y si avanzamos actualizamos los tiles...¡FALTA!
       If (region_act<>ultima_region)
            From fila= 0 To 11;
               columna=region_act+19;
              map_put(0,0,tiles[(fila*20)+columna],(region_act-1)*64,fila*64);
            End
            ultima_Region=region_act;
       End
       Frame;
    End
End

Process nave()
Begin
    x=100;
    y=200;
    graph=new_map(20,20,32);map_clear(0,graph,rgb(0,0,0));
    loop
        If (key(_right)) x=x+2; End
        If (key(_left)) x=x-2; End
        If (key(_down)) y=y+2; End
        If (key(_up)) y=y-2; End
        frame;
```

```
    end
end
```

El código que nos interesa está en el proceso “mapa_tileado()”. Lo primero que hacemos es incrustar dentro del gráfico de fondo –transparente- los distintos tiles uno al lado del otro, a partir de la función **map_put()**, recorriendo convenientemente tanto las posiciones dentro de la pantalla donde se colocarán los tiles, como los elementos de la tabla donde está definido el tile que queremos visualizar. Y a partir de entonces,se entra en un Loop donde si la nave avanza los tiles deberían posicionarse convenientemente. No obstante, esta parte del código no funciona todavía porque no se ha implementado

Pero no te quedarás sin las ganas de comprobar cómo se hace un escenario tileado y además scrolleable. Nos tendremos que olvidar para siempre de utilizar **start_scroll()**, ya que el scroll por tiles requiere que todo el proceso de intercambio de baldosas cuando un personaje-camara se mueve por la pantalla se programe "a mano". No deja de ser un tema relativamente avanzado que nosotros sólo tocaremos de puntillas. El siguiente ejemplo es muy completo (recomiendo leer los comentarios), porque además de implementar un scroll tileado, también incorpora un editor de niveles, de manera que se puedan colocar tiles a modo de muro que el personaje no pueda traspasar, e incluso guardar estos niveles. De hecho, a partir de esta idea, otro interesante ejercicio sería construir un completo editor de paisajes tileados a partir de diferentes muestras de tiles, las cuales se podrían colocar incrustados en algún tipo de rejilla (mediante el cursor del ratón, por ejemplo) que daría forma al paisaje.

```
import "mod_key";
import "mod_proc";
import "mod_text";
import "mod_screen";
import "mod_draw";
import "mod_map";
import "mod_file";

const
          TILESIZE = 50;
          DIM = 50; //Dimensiones del nivel(DIM*DIM)
          SCREENX = 320;
          SCREENY = 240;
end

global
          //Estrutura para guardar los datos del nivel
          struct level
            int tile[DIM][DIM]; //Guarda qué tiles ("normales" ó "muros") están dónde
            int start[1]; //Posición de inicio de nuestro personaje
          end

/*Estructura para scrollear el nivel. sx y sy guardan el punto de la "cámara" en pixeles,
respecto la esquina superior izquierda*/
          struct scrol
             int sx;
             int sy;
             int cam;
          end

          int editormode; //true si se está en modo editor, false si no

          int __A = _control; //Teclas
          int __L = _backspace;
          int __R = _tab;

          int graftile1, graftile2;

local
          int xs,ys;
end

process main()
begin
set_mode(SCREENX,SCREENY,32);
```

```
//El cuadrado gris será el tile "normal" por donde podrá moverse nuestro personaje
graftile1=new_map(50,50,32);map_clear(0,graftile1,rgb(100,100,100));
//El cuadrado rojo será el tile "muro", que nuestro personaje no podrá traspasar
graftile2=new_map(50,50,32);map_clear(0,graftile2,rgb(255,0,0));

//Si el nivel existe se carga, si no se crea uno nuevo
if(exists("level.lvl"))
  load("level.lvl",level);
else
  //Se rellena el nivel con el tile "normal"
  for(x=0;x<DIM;x++)
    for(y=0;y<DIM;y++)
      level.tile[x][y] = graftile1;
    end
  end
  //La posición de inicio por defecto de nuestro personaje será el centro del nivel
  level.start[0] = DIM/2;
  level.start[1] = DIM/2;
end

editor();

loop
  if(key(_esc)) exit();end
  frame;
end
end


//Proceso Editor. Permite al usuario cambiar los tiles del nivel como desee
Process Editor()
private
          int tilex;
          int tiley;
          int goodToMove;
end
begin
RenderLevel(); //Dibuja el nivel
editormode = 1;
tilex = DIM/2;
tiley = DIM/2;

write(0,0,0,0,"Hay dos tiles disponibles: 'normal' y 'muro'");
write(0,0,10,0,"Pulsa CTRL para cambiar el tile");
write(0,0,20,0,"Pulsa BACKSPACE para guardar el nivel");
write(0,0,30,0,"Pulsa TAB para volver al modo jugar");
write(0,0,40,0,"PUlsa ENTER para establecer punto de inicio");

//Especifica un cuadrado negro como el gráfico de este proceso
graph=new_map(50,50,32);
drawing_map(0,graph);drawing_color(rgb(0,0,0));draw_rect(0,0,49,49);
scrol.cam = id; //Hace que la cámara diga al cuadrado rojo

loop
  if(goodToMove !=0) //Navega el "cursor" a través del nivel
    if(key(_left) and tilex>0) tilex--; end
    if(key(_right) and tilex<DIM-1) tilex++; end
    if(key(_up) and tiley>0) tiley--; end
    if(key(_down) and tiley<DIM-1) tiley++; end
  end

  if(!key(_left) and !key(_right) and !key(_up) and !key(_down))
    goodToMove = true;
  else
    goodToMove = false;
  end

  //Cambia los tiles (sólo hay 2 disponibles).
```

```
  if(key(__A))
    if(level.tile[tilex][tiley] == graftile1)
          level.tile[tilex][tiley] = graftile2;
    else
          level.tile[tilex][tiley] = graftile1;
    end
    while(key(__A))frame;end
  end

  //Cambia el punto de inicio del jugador
  if(key(_enter))
    level.start[0] = tilex;
    level.start[1] = tiley;
    while(key(_enter))frame;end
  end

  //Calcula la posición con la posición de los tiles
  xs = tilex * TILESIZE + TILESIZE/2;
  ys = tiley * TILESIZE + TILESIZE/2;

  //Graba el nivel
  if(key(__L))
    save("level.lvl",level);
    while(key(__L))frame;end
  end

  //Cambia al modo "jugar"
  if(key(__R))
    while(key(__R))frame;end
    delete_text(all_text);
    character();
    return;
  end
  frame;
end
end


//Renderiza ("dibuja") el nivel en la pantalla a cada fotograma
Process RenderLevel()
private
          int tleft,tright,tup,tdown; //Número de tile en los bordes del área visible
          int idgraf; //Gráfico (círculo azul) que marca el punto de inicio del jugador
end
begin
idgraf=new_map(50,50,32);
drawing_map(0,idgraf);drawing_color(rgb(0,0,255));draw_fcircle(25,25,25);
loop
  clear_screen();

  if(exists(scrol.cam)) //Sigue un proceso cámara si alguno está definido
    scrol.sx = scrol.cam.xs - SCREENX/2;
    scrol.sy = scrol.cam.ys - SCREENY/2;
  end

  //Se asegura que la posición de la cámara no salga fuera del nivel
  if(scrol.sx < 0)scrol.sx = 0;end
  if(scrol.sy < 0)scrol.sy = 0;end
  if(scrol.sx > DIM * TILESIZE - SCREENX)scrol.sx = DIM * TILESIZE - SCREENX;end
  if(scrol.sy > DIM * TILESIZE - SCREENY)scrol.sy = DIM * TILESIZE - SCREENY;end

  /*Establece las coordenadas reales del proceso cámara (sólo es una buena solución
  no hay múltiples procesos que necesitan que sean convertidas sus coordenadas)*/
  if(exists(scrol.cam))
    scrol.cam.x = scrol.cam.xs - scrol.sx;
    scrol.cam.y = scrol.cam.ys - scrol.sy;
  end
```

```
  //Determina qué tiles están en pantalla y deberían ser dibujadas
  tleft = scrol.sx / TILESIZE - 1;
  tright = tleft + SCREENX / TILESIZE + 2;
  tup = scrol.sy / TILESIZE - 1;
  tdown = tup + SCREENY / TILESIZE + 2;

  //Chequea si permanecen dentro de los bordes del nivel
  if(tleft < 0)tleft = 0;end
  if(tup < 0)tup = 0;end
  if(tright >= DIM)tright = DIM -1;end
  if(tdown >= DIM)tdown = DIM -1;end

  //Y dibuja el nivel resultante
  for(x=tleft;x=<tright;x++)
    for(y=tup;y=<tdown;y++)
put(0,level.tile[x][y],x * TILESIZE - scrol.sx + TILESIZE/2,y * TILESIZE - scrol.sy +
TILESIZE/2);
    end
  end

  //Dibuja un círculo rojo representando el punto de inicio del jugador, si se está en el
modo editor
  if(editormode == 1)
put(0,idgraf,level.start[0] * TILESIZE - scrol.sx + TILESIZE/2,level.start[1] * TILESIZE
- scrol.sy + TILESIZE/2);
  end

  frame;
end
end



/*Personaje que se mueve a través del nivel creado con el editor, colisionando con los
tiles "muro".Se podría implementar una animación del tipo caminar con varios gráficos,
pero para simplificar el ejemplo, nuestro personaje tan sólo tendrá un gráfico
invariable*/

Process Character()
private
          int radius = 20; //El radio del personaje
          int idperso; //Identificador del gráfico del personaje
end
begin
editormode = 0; //Establece editormode a falso, es decir, no dibuja el nodo de inicio del
jugador

graph=new_map(30,30,32);map_clear(0,graph,rgb(0,255,0));

//Coloco el personaje en la posición de inicio
xs = level.start[0] * TILESIZE + TILESIZE/2;
ys = level.start[1] * TILESIZE + TILESIZE/2;

//Asigno la cámara de nuestro "scroll" hecho a mano al personaje, para que éste sea
perseguido
scrol.cam = id;

write(0,0,10,0,"Pulsa TAB para volver al modo editor");

loop
/*Mueve el personaje utilizando las coordenadas xs y ys, ya que éstas son automáticamente
convertidas en coordenadas de pantalla por el proceso "renderlevel()"*/
  if(key(_up))
     xs = xs + get_distx(angle,8);//Se desplaza 8 pixeles
     ys = ys + get_disty(angle,8);
  end
  if(key(_down))
    xs = xs + get_distx(angle,-8);
```

```
    ys = ys + get_disty(angle,-8);
  end
  if(key(_right)) angle = angle -5000; end
  if(key(_left))  angle = angle +5000; end

  //Cambia al modo editor
  if(key(__R))
    while(key(__R))frame;end //Se espera a que se suelte la tecla
    delete_text(all_text);
    editor();
    return; //Mata el proceso actual
  end

  //Detección de colisiones para los 4 costados utilizando el radio dado
  while(level.tile[(xs+radius)/TILESIZE][ys/TILESIZE] == graftile2) xs--; end
  while(level.tile[(xs-radius)/TILESIZE][ys/TILESIZE] == graftile2) xs++; end
  while(level.tile[xs/TILESIZE][(ys+radius)/TILESIZE] == graftile2) ys--; end
  while(level.tile[xs/TILESIZE][(ys-radius)/TILESIZE] == graftile2) ys++; end
  frame;
end
end
```

Pasando ahora a otra aplicación de los mapas tileados, puede ser que en algún juego, en vez de interesarnos que el personaje se mueva por un escenario tileado tal como acabamos de ver, nos interese que sea el propio escenario el que se pueda mover -manteniendo fijos todos sus elementos-, a modo de rastreo, (como simulando la acción de una exploración de territorio), mediante la acción de los cursores, por ejemplo. He aquí un ejemplo de código que permite este efecto de movimiento de un escenario tileado completo. Para poder probarlo, necesitarás tener el mismo FPG que creamos en el ejemplo anterior, ya que utilizaremos las 11 baldosas de 64x64 que hay contenidas allí.. El escenario (más grande que la pantalla del juego para poder experimentar el efecto, claro) constará de 20 filas compuestas por 25 tiles cada una . El truco que usaremos para rellenar el escenario es idéntico al del ejemplo anterior: usaremos una matriz donde cada elemento identificará al número de imagen dentro del fpg que corresponda al dibujo que se quiera ver en su posición dada dentro de la matriz. La diferencia con el ejemplo anterior viene en el proceso "ponmapa()", que es el que se encarga de mover los tiles de acuerdo al movimiento del cursor, obteniendo así el efecto deseado.

```
import "mod_text";
import "mod_key";
import "mod_map";
import "mod_video";
import "mod_proc";
import "mod_screen";


Global
/*Esta matriz debería tener 25X20=500 elementos, cuyos valores serán números de imágenes
dentro del fpg utilizado*/
    Byte lectura[]=
    1,1,2,10,11,4,1,1,1,1,2,3,9,3,8,3,3,3,4,2,9,8,7,6,5,
    3,3,3,10,11,3,3,3,3,4,2,3,9,3,8,3,3,3,4,2,4,3,2,1,10,
    3,3,3,10,11,3,3,3,3,4,2,3,9,3,8,5,6,6,7,9,11,10,9,8,7,
    1,1,2,10,11,3,3,3,3,3,3,3,9,3,8,5,6,6,7,9,6,5,4,3,2,1,
    3,3,3,10,11,3,3,3,3,4,2,3,9,3,8,5,6,6,7,9,11,10,9,8,7,
    1,1,2,10,11,5,6,6,6,7,9,3,9,3,8,5,6,6,7,9,11,10,11,10,1,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,9,9,9,3,4,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,9,9,6,6,7,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,8,8,2,3,5,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,9,3,9,3,9,8,7,6,5,4,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3,6,7,5,4,8,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3,2,1,3,4,6,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3,9,8,7,6,5,
    3,3,3,10,11,3,3,3,3,4,2,3,9,3,8,3,3,3,4,2,4,3,2,1,10,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,9,3,9,3,9,8,7,6,5,4,
    1,1,2,10,11,3,3,3,3,3,3,3,9,3,8,5,6,6,7,9,6,5,4,3,2,1,
    3,3,3,10,11,9,9,3,3,3,9,3,9,3,8,3,3,3,3,3,6,7,5,4,8,
    1,1,2,10,11,5,6,6,6,7,9,3,9,3,8,5,6,6,7,9,11,10,11,10,1,
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,9,9,9,3,4,
```

```
    3,3,3,10,11,5,6,6,6,7,9,3,9,3,8,3,10,11,3,3,9,9,6,6,7;
end


Process Main()
Private
    int mx;
    int my;
End
Begin
          set_mode(640,480);
          load_fpg("tiles.fpg");

          ponmapa(0,0);
          write(0,20,20,0,"Cursores para mover");
          While (NOT key (_esc))
                    If (key(_left) AND mx>0)
                              mx--;
                              ponmapa(mx,my);
                    End
                    If (key(_right) AND mx<20)
                              mx++;
                              ponmapa(mx,my);
                    End
                    If (key(_down) AND my<10)
                              my++;
                              ponmapa(mx,my);
                    End
                    If (key(_up) AND my>0)
                              my--;
                              ponmapa(mx,my);
                    End
                    Frame;
          End
End

Process ponmapa(x,y)
private
    int xx;
    int yy;
end
Begin
/*La idea de estos bucles y la función put() es idéntica al del ejemplo anterior. Lo
único que cambia es que ahora interviene también la x e y del propio proceso.*/
          From xx=0 To 24;
                    From yy=0 To 19;
/*El +32 es porque el punto de control 0 esta en el centro.Así los tiles de la primera
columna no se verán partidos por la mitad*/
                              put(0,lectura[((y+yy)*25)+(x+xx)],xx*64+32,yy*64+32);
                    End
          End
End
```

Si deseas saber más sobre el funcionamiento de los tiles y cómo pintarlos en un escenario scrolleable, puedes consultar las interesantísimas páginas (en inglés) http://gpwiki.org/index.php/RPG_CHAR ó http://gpwiki.org/index.php/RPG_Map (y su continuación http://gpwiki.org/index.php/RPG_MAP2)

El siguiente ejemplo es un programa que nos permitirá, a partir de una imagen dada (en nuestro caso, llamada "tira.png",pero su nombre se podría haber introducido como parámetro de la línea de comandos), trocearla en un conjunto de tiles 20x20 gracias a la función **map_block_copy()**. Tal como está escrito el código, la imagen sólo podrá tener un tamaño máximo de 320x220 (y en todo caso múltiple de 20 en ambas coordenadas), debido al tamaño de la tabla *tiles[]*, pero esto es fácilmente modificable: en nuestro ejemplo la imagen ha de ser de 320x220 exactos. Estos tiles además serán grabados en el disco duro, dentro de la carpeta "mistiles" (¡ha de existir previamente!) situada en nuestra carpeta actual, con un nombre dado por la posición que ocupa dicho tile dentro de la serie.

Además, para comprobar que efectivamente el troceado se ha realizado correctamente, este ejemplo

realiza el paso contrario también: a partir de la matriz de tiles (ya cargada en memoria en el paso anterior) se encarga de generar una imagen final, que la mostrará por pantalla y que deberá coincidir con la imagen original troceada.

```
import "mod_text";
import "mod_key";
import "mod_map";
import "mod_video";
import "mod_proc";
import "mod_screen";

const
          ancho_tile=20;
          alto_tile=20;
/*Resultado de tener 11 tiles horizontales como mucho. Puede ser un valor menor, pero
siempre múltiple de ancho_tile*/
          ancho_mapa=220;
/*Resultado de tener 16 tiles verticales como mucho. Puede ser un valor menor, pero
siempre múltiple de alto_tile*/
          alto_mapa=320;
end

process main()
private
          int tile; //numero de tile
          int i,j;
          int num_tiles_horiz, num_tiles_vert;
          int img_original,img_final;
/*Tabla de 11x16=176 elementos, cada uno de los cuales es 20x20 (tal como se indica en la
sección de constantes).*/
          int tiles[176];
end
begin
set_mode(280,20,32);

//cojer tiles
img_original=load_png("tira.png");
num_tiles_horiz=map_info(0,img_original,g_width)/ancho_tile;
num_tiles_vert=map_info(0,img_original,g_height)/alto_tile;

write(0,0,0,0,"Se procederá a trocear la imagen");
write(0,0,10,0,"Pulse SPACE para continuar.");
from j=0 to num_tiles_vert-1;
          from i=0 to num_tiles_horiz-1;
                    tiles[tile]=new_map(ancho_tile,alto_tile,32);
/*Esta función copia a partir de la posición (0,0) de tiles[tile] un trozo de
img_original de igual tamaño que el tile, a partir de la posición i*ancho y j*alto del
tile en cuestión*/
map_block_copy(0,tiles[tile],0,0,img_original,i*ancho_tile,j*alto_tile,ancho_tile,alto_ti
le,0);
                    save_png(0,tiles[tile], "mistiles/" +j+i+".png");
                    tile++;
          end
end
while(not key(_space)) frame; end
delete_text(0);
//fin cojer tiles


//poner tiles
write(0,0,0,0,"Se procederá a recomponer la imagen");
write(0,0,10,0,"Pulse ENTER para mostrarla por pantalla");
img_final=new_map(ancho_mapa,alto_mapa,32);
num_tiles_horiz=ancho_mapa/ancho_tile;
num_tiles_vert=alto_mapa/alto_tile;
tile=0;
from j=0 to num_tiles_vert-1;
          from i=0 to num_tiles_horiz-1;
/*Lo de sumar ancho_tile/2 y alto_tile/2 es por la razón, ya comentada, de que los mapas
```

```
suelen tener el punto de control 0 en el centro del mapa (por eso si no se sumara esta
cantidad, el primer tile se vería recortado)*/
     map_put(0,img_final,tiles[tile],i*ancho_tile+ancho_tile/2,j*alto_tile+alto_tile/2);
                    tile++;
          end
end
while(not key(_enter)) frame; end
delete_text(0);
//fin poner tiles

//mostrar resultado
set_mode(ancho_mapa,alto_mapa,32);
graph=img_final;
x=ancho_mapa/2;
y=alto_mapa/2;
while(not key(_esc)) frame; end
end
```

Aplicando la misma filosofía que en código anterior, podríamos incluso generarnos nuestros propios procesos troceadores. Por ejemplo: el siguiente código implementa un proceso que troceará siempre la imagen pasada por parámetro en 9 tiles (3x3):

```
import "mod_map";

process main()
private
          int idpng;
end
begin
          idpng=load_png("tira.png");
          trocea3x3(0,idpng);
end

process trocea3x3(file, graph)
private
          int g,w,h;
          int i,j;
          int tile[8];
end
begin
          g = map_clone(file, graph);
          w=map_info(0, g, G_WIDTH)/3;
          h=map_info(0, g, G_HEIGHT)/3;
          for (i=0;i<3;i++)
                    for (j=0;j<3;j++)
                              tile[i*3+j] = new_map(w,h,32);
                              map_block_copy(0,tile[i*3+j],0,0,g,i*w,j*w,w,h,0);
                              save_png(0,tile[i*3+j], "mistiles/" +i+j+".png");
                    end
          end
          unload_map(0,g);
end
```

Incluso podríamos trocear una imagen y almacenar los tiles resultantes dentro de un FPG, tal como hace le siguiente código:

```
import "mod_key";
import "mod_map";
import "mod_video";
import "mod_proc";
import "mod_draw";

process main()
private
          int tilesize=16;
          int i,j;
          int efepege;
```

```
          int tira;
          int grafico=1;
          int color;
          int numtileshoriz, numtilesvert;
end
Begin
tira=load_png("tira.png");
efepege=fpg_new();
numtilesvert=map_info(0,tira,g_height)/tilesize;
numtileshoriz=map_info(0,tira,g_width)/tilesize;
graph=new_map(tilesize,tilesize,32);
from j=0 to numtilesvert;
     from i=0 to numtileshoriz;
          from y=j*tilesize to (j+1)*tilesize;
                    from x=i*tilesize to (i+1)*tilesize;
                       color=map_get_pixel(0,tira,x,y);
                       map_put_pixel(efepege,graph,x-(i*tilesize),y-(j*tilesize),color);
                    end
          end
     fpg_add(efepege,grafico,0,graph);
     grafico++;
     end
end
save_fpg(efepege,"output.fpg");
End
```

El siguiente código lleva un paso más allá la idea de trocear una imagen en tiles. En este caso, a partir de un fichero PNG especificado (en nuestro caso llamado "fondo.png"), el siguiente ejemplo generará otro fichero PNG llamado "tiles.png" compuesto por una tira con todos los tiles DIFERENTES (dispuestos uno detrás de otro) que se han generado al trocear la imagen. El tamaño de éstos se puede cambiar para obtener una tira diferente cada vez. Además, por pantalla se visualiza la división que se realiza de la imagen para poder observar si es la que nos interesa.

```
Import "mod_key";
Import "mod_proc";
Import "mod_video";
Import "mod_screen";
Import "mod_map";
Import "mod_draw";
Import "mod_mem";

Global
          int bg_width,bg_height;
          int graphs[100];
          int len_graphs;
          int TILE_WIDTH = 16;
          int TILE_HEIGHT= 16;
end

process main()
Begin
          set_mode(800,600,32);
          backgr("fondo.png");
          while (!key(_esc))
                    frame;
          end
          let_me_alone();
end

process backgr(string foto)
private
          int id_linereg;
          int temp_graph;
          int tile_data[100][100];
          int w_tile,h_tile;
          int i,j,posicion;
end
begin
```

```
           graph = load_png(foto);
           bg_width=map_info(0,graph,g_width);
           bg_height=map_info(0,graph,g_height);
           x=bg_width/2;
           y=bg_height/2;
           id_linereg=linereg(x,y);
           while (!key(_esc))
              if (key(_s))
                     memset(&tile_data,0,sizeof(tile_data));
                     for (i=0;i<bg_width;i=i+TILE_WIDTH)
                              for (j=0;j<bg_height;j=j+TILE_HEIGHT)
                                       temp_graph= new_map(TILE_WIDTH,TILE_HEIGHT,32);
                     map_block_copy(0,temp_graph,0,0,graph,i,j,TILE_WIDTH,TILE_HEIGHT,0);
                                       posicion = pos_en_mapa(temp_graph);
                                       if (posicion==-1)
                                                graphs[len_graphs] = temp_graph;
                                                posicion = len_graphs;
                                                len_graphs++;
                                       end
                                       tile_data[i/TILE_WIDTH],j/TILE_HEIGHT]=posicion;
                              end
                     end
                     w_tile = i/TILE_WIDTH;
                     h_tile = j/TILE_HEIGHT;
                     temp_graph=new_map(len_graphs*TILE_WIDTH,TILE_HEIGHT,32);
                     for (i=0;i<len_graphs;i++)
map_block_copy(0,temp_graph,i*TILE_WIDTH,0,graphs[i],0,0,TILE_WIDTH,TILE_HEIGHT,0);
                     end
                     save_png(0,temp_graph,"tiles.png");
                     while (key(_s)) frame;end
              end
              frame;
           end
           unload_map(0,graph);
end

process linereg(x,y)
begin
           z=-1;
           graph = draw_linereg(new_map(bg_width,bg_height,32));
           alpha = 80;
           while (!key(_esc))
                     frame;
           end
           unload_map(0,graph);
end


function draw_linereg(graph)
private
           int i,j;
begin
           drawing_alpha(255);
           drawing_map(0,graph);
           map_clear(0,graph,RGB(0,0,0));
           drawing_color(RGB(255,0,255));
           for (i=0;i<bg_width;i=i+TILE_WIDTH)
                     for (j=0;j<bg_height;j=j+TILE_HEIGHT)
                              draw_rect(i,j,i+TILE_WIDTH-2,j+TILE_HEIGHT-2);
                     end
           end
           return graph;
end

function es_igual(int g1,int g2)
private
           int i_gr,j_gr;
end
```

```
begin
          for (i_gr=0;i_gr<TILE_WIDTH;i_gr++)
             for (j_gr=0;j_gr<TILE_HEIGHT;j_gr++)
                      if (map_get_pixel(0,g1,i_gr,j_gr) != map_get_pixel(0,g2,i_gr,j_gr))
                               return 0;
                      end
             end
          end
          return 1;
end

function pos_en_mapa(int graph1)
private
          int i;
end
begin
          for (i=0;i<len_graphs;i++)
                    if (es_igual (graph1,graphs[i]))
                               return i;
                    end
          end
          return -1;
end
```

Otra manera diferente de utilizar los tiles es la siguiente: se puede tener un mapa (en escala reducida) que haga de "plantilla" para poder construir a partir de él el mapa visible formado por los tiles . Es decir, el mapa "plantilla" sería similar a un mapa de durezas: cada píxel tendría un color determinado que indicaría que en esa posición (corregida la escala) se colocaría un tile determinado. Así, se podría tener por ejemplo asociado el color amarillo del mapa "plantilla" a un tile que fuera un muro, el color rojo a un tile que fuera un árbol, etc. De esta manera, se podría mantener un conjunto reducido de tiles pero usando varios mapas "plantillas" diferentes, de manera que se podrían construir y reutilizar escenarios muy fácilmente con tan sólo cambiando el mapa "plantilla". Vamos a verlo con un ejemplo:

```
Import "mod_video";
Import "mod_key";
Import "mod_map";
Import "mod_draw";
Import "mod_text";
Import "mod_screen";

Global
          int mapplantilla;
          int mapescenario;
End

Process Main()
begin
          set_mode(320,240,32);
          crearPlantilla(); //La plantilla puede ser un simple gráfico ya existente
//        put_screen(0,mapplantilla);
          pintarEscenario(); //A partir de la plantilla cargada previamente
          put_screen(0,mapescenario);
          repeat
                    frame;
          until(key(_esc))
end


/*El mapa "plantilla" tiene una anchura y altura que es la mitad del escenario visible.
Por lo tanto, el factor escala hay que reducirlo o aumentarlo (según sea necesario) x2.
Esto incluye el tamaño de los tiles, que en la plantilla tienen un tamaño de 20x20, pero
en el escenario serán de 40x40*/
Function crearPlantilla()
Private
   int mapcamino,mapmuro,maparbol,maphierba;
   int fila,columna;
   int anchoplant=160;
   int altoplant=120;
```

```
End
Begin
//Genero los gráficos que van a constituir la plantilla, y ésta misma
   mapplantilla=new_map(anchoplant,altoplant,32);
   mapcamino=new_map(20,20,32);map_clear(0,mapcamino,rgb(255,255,0));
   mapmuro=new_map(20,20,32);map_clear(0,mapmuro,rgb(255,0,0));
   maparbol=new_map(20,20,32);map_clear(0,maparbol,rgb(0,0,255));
   maphierba=new_map(20,20,32);map_clear(0,maphierba,rgb(0,255,0));

/*Genero la plantilla propiamente, ubicando de alguna manera los distintos gráficos.Se
podría tener la plantilla ya hecha como gráfico aparte, pero en este ejemplo se generará
dinámica de forma tileada un poco a lo bruto. La idea sería tener diferentes plantillas
(por ejemplo, con diferentes funciones crearPlantillaX()), o bien con diferentes gráficos
ya creados, y utilizar una u otra según convenga.*/
    From fila= 0 To 15; //En un ancho de 160px caben 16 tiles de 20px de ancho
               From columna= 0 To 11;//En un alto de 120px caben 12 tiles de 20px de alto
                                map_put(0,mapplantilla,maphierba,columna*20,fila*20);
                           if (columna==3 and fila==4)
                                map_put(0,mapplantilla,maparbol,columna*20,fila*20);
                           elif (columna==7)
                                map_put(0,mapplantilla,mapmuro,columna*20,fila*20);
                           elif (fila==4)
                                map_put(0,mapplantilla,mapcamino,columna*20,fila*20);
                           end
               End
    End
    return mapplantilla;
End

Function pintarEscenario()
Private
   int tilecamino,tilemuro,tilearbol,tilehierba;
   int fila,columna;
   int anchoplant=160;
   int altoplant=120;
   int colorplantilla;
   int amarillo,rojo,azul,verde;
End
begin
//Genero los tiles que se irán colocando en el escenario según lo que marque la
plantilla, y el propio escenario
   mapescenario=new_map(320,240,32);
   tilecamino=new_map(40,40,32);drawing_map(0,tilecamino);
   drawing_color(rgb(255,255,0));draw_fcircle(20,20,20);
   tilemuro=new_map(40,40,32);drawing_map(0,tilemuro);
   drawing_color(rgb(255,0,0));draw_fcircle(20,20,20);
   tilearbol=new_map(40,40,32);drawing_map(0,tilearbol);
   drawing_color(rgb(0,0,255));draw_fcircle(20,20,20);
   tilehierba=new_map(40,40,32);drawing_map(0,tilehierba);
   drawing_color(rgb(0,255,0));draw_fcircle(20,20,20);

//Coloco propiamente los tiles
    amarillo=rgb(255,255,0);
    rojo=rgb(255,0,0);
    azul=rgb(0,0,255);
    verde=rgb(0,255,0);
    From fila= 0 To 15;
               From columna= 0 To 11;
                         colorplantilla=map_get_pixel(0,mapplantilla,columna*20,fila*20);
                        switch(colorplantilla)
                          case amarillo:
                             map_put(0,mapescenario,tilecamino,columna*40+20,fila*40+20);
                          end
                          case rojo:
                             map_put(0,mapescenario,tilemuro,columna*40+20,fila*40+20);
                          end
                          case azul:
                             map_put(0,mapescenario,tilearbol,columna*40+20,fila*40+20);
```

```
                        end
                        case verde:
                           map_put(0,mapescenario,tilehierba,columna*40+20,fila*40+20);
                        end
                     end
            End
    End
    return mapescenario;
end
```

Existen programas "tileadores profesionales", es decir, especialmente pensados para generar tiles a la carta a partir de imágenes dadas, y mucho más. Uno de ellos es el Tile Studio, libre pero sólo para Windows: http://tilestudio.sourceforge.net . Incluye un editor de mapa de bits y un diseñador de niveles.

Si se quiere profundizar mucho más en el desarrollo de juegos tileados, recomiendo muy fervientemente el espléndido tutorial "Tile Based Games", disponible en http://www.tonypa.pri.ee/tbw/start.html. También es interesante (aunque esté orientada al lenguaje Flash) la página http://oos.moxiecode.com/

## *Conceptos de física newtoniana

Hasta ahora los procesos de los juegos que hemos hecho poseen movimientos que en la vida real son poco frecuentes: movimientos rectilíneos, sin aceleración, sin rozamiento,etc. Imagínate por un momento que quisieras hacer un juego de coches: para dar realismo deberías poder hacer que derrapara, que volcara, que en un choque saliera despedido (o no)…todo eso ahora mismo no lo sabemos hacer. Y no porque no sepamos más Bennu; no se trata de eso. Se trata de incorporar al código de nuestro juego las fórmulas físicas necesarias que rigen el movimiento en la vida real,de tal manera que nuestros procesos emulen los comportamientos que nosotros observamos día a día: gravedad, elasticidad, acción-reacción,etc. Es decir, deberemos de incorporar al movimiento de los procesos ecuaciones del movimiento newtoniano. Si lo logras, verás como el comportamiento de los móviles de tu juego será increiblemente más real y creíble.

Evidentemente, los efectos que podamos conseguir son tantos como casos particulares que tiene la física. Pero para empezar, de momento solamente estudiaremos un caso particular: **el tiro parabólico**. La idea es que en nuestro juego podamos tener un cañón, el cual, dependiendo de su inclinación respecto el suelo, pueda lanzar bombas más o menos lejos y más o menos alto. Primero hay que conocer, no obstante la parte teórica del asunto, para comprender lo que queremos conseguir. Al estudiar cualquier movimiento en dos dimensiones siempre se descompone en sus componentes X e Y, ya que así es más fácil estudiarlo. En cualquier manual de física se puede ver que la componente X del movimiento parabólico es un movimiento de velocidad horizontal constante, de fórmula:

$$x = x_0 + v_{0x} \cdot t$$

donde las variables en esta fórmula son las siguientes:

"**x**": es la coordenada X actual
"$x_0$": la coordenada X de origen (desde donde se dispara)
"$v_{0x}$": la velocidad horizontal, que es constante
"**t**": el tiempo transcurrido en cada instante (ya sea en fotogramas, segundos, o lo que quieras, pero si no tienes muy claro el tema, en Bennu mejor usar fotogramas), de modo que en nuestro caso "t" sería el fotograma actual de la animación del movimiento parabólico.

Además de ese movimiento, tenemos la componente Y, que es un movimiento vertical de velocidad variable (la bomba sube y baja con cierta aceleración o desaceleración), cuya fórmula es:

$$y = y_0 + v_{0y} \cdot t + 0.5 \cdot g \cdot t^2$$

donde las variables son:

"**y**": la coordenada Y actual

"$y_0$": la coordenada Y de la que parte el disparo
"$v_{0y}$": la velocidad de partida vertical
"**t**": el tiempo transcurrido, es el mismo para Y y para X
"**g**": valor de la gravedad

Por tanto el movimiento parabólico se compone de la suma de estos dos movimientos: un movimiento horizontal (rectilíneo y de velocidad constante) y un movimiento vertical (rectilíneo acelerado por el campo gravitatorio terrestre).Ok, ya tenemos la base teórica para crear nuestro movimiento parabólico. Ahora nos toca aplicar todo a nuestro juego.

Primero tenemos que decidir cuánto tiempo (en fotogramas) durará la animación, para calcular las velocidades necesarias, "$v_{0x}$" y "$v_{0y}$". El razonamiento para decidir esto sería, por ejemplo, éste: si mi juego va a unos 30 fps, y quiero que la bala esté 2 segundos en el aire antes de golpear el objetivo (o el suelo, si la esquivan), el valor máximo que podrá tomar la variable "**t**" deberá ser de 60 fotogramas. A partir de aquí, toca calcular el resto de variables para que esto sea así. Empecemos con la X.

El movimiento horizondal de la bala venía dado por la fórmula escrita en el primer cuadro. Para t=60 (que es el tiempo en que alcanza su objetivo, hemos dicho), la bala se encontrará en la coordenada X del objetivo, por lo que "**x**" será igual a la coordenada de la víctima. Y también conocemos la coordenada "$x_0$" de la que sale la bala, ya que es la misma que la del enemigo que la dispara. Pongamos que en el momento del disparo la x del enemigo que nos está disparando es 300, y la del protagonista es 20. Sustituimos los datos que tenemos en la fórmula inicial: $20=300+v_{0x}\cdot 60$ Despejamos en la fórmula y nos queda: $v_{0x}=(20-300)/60$ , lo que equivale a $v_{0x}=(x_0-x)/t$

Así calculamos la velocidad horizontal necesaria para que la bala recorra en 60 fotogramas la distancia que hay entre el enemigo y el protagonista (en el momento de lanzar la bala). La velocidad toma un valor negativo, esto se debe a que su movimiento será hacia la izquierda.

Bien, ya tenemos solucionado el movimiento horizontal, ahora nos queda el vertical. Partamos de la fórmula del segundo cuadro. Sabemos que t=60, porque para que el movimiento cuadre las "**t**" deben ser iguales en ambos movimientos, evidentemente. Sabemos "$y_0$", ya que es la coordenada Y del enemigo que dispara, y sabemos el valor "**y**" necesario para calcular la velocidad vertical inicial ("$v_{0y}$"), ya que "**y**" debe ser la Y del proceso que recibe el disparo. También sabemos "**g**", la gravedad, en la Tierra toma un valor de aproximadamente 9.8, pero en nuestro juego podemos poner el que pensemos que queda mejor para la jugabilidad del mismo. Ya solo nos queda averiguar "$v_{0y}$", y para ello despejamos: $v_{0y} = (y - y_0 - 0.5 \cdot g \cdot t^2)/ t$

Si la velocidad inicial (llamémosle "**$v_0$**") la conoces de entrada –porque te interesa por ejemplo que sea una en concreto siempre- , no hace falta hacer estos cálculos para lograr obtener sus componentes horizontal y vertical.Para un ángulo inicial de tiro, las componentes de la velocidad inicial "$v_{0x}$" y "$v_{0y}$" vienen dadas respectivamente por $cos(angulo\_inicial)\cdot v_0$ y $sen(angulo\_inicial)\cdot v_0$ , siendo, ya he dicho, "**$v_0$**" la velocidad inicial que se le da al proyectil.

Ok, como ya tenemos de una u otra manera las velocidades necesarias para ambos movimientos, ya podemos construir nuestro proceso. Supongamos que *prota* es la variable que almacena el id del protagonista (ej: *prota =protagonista(x,y);*) Y que el proceso "enemigo()" es el que crea la bala:

```
PROCESS bala()
PRIVATE
           int y0;
           int x0;
           int y_final;
           int x_final;
           int v0x;
           int voy;
           int vy; //velocidad Y actual
           float g;
           float gravedad_por_frame;
           int t;
END
BEGIN

           x = father.x;
           y = father.y;
```

```
          t=60;
          g=9.8;
          v0x = (prota.x -x)/t; //calculamos la velocidad horizontal
          v0y = (prota.y - y0 - 0.5*g*t*t) / t; //calculamos la velocidad vertical
          vy = v0y;//La velocidadY al principio es igual que la inicial

/*IMPORTANTE: La gravedad vale 9.8 segundos, pero nosotros las fórmulas las estamos
calculando tomando como unidad de tiempo el fotograma. Así que hay que transformar la
gravedad en unidades de fotograma.*/
           gravedad_por_frame = g/t;  //Ha de ser un valor constante

/*La clave del asunto.En cada fotograma aumentamos una cantidad constante ("v0x") la
posición horizontal de la bala y aumentamos una cantidad variable ("vy") -que a su vez
varía una cantidad fija "gravedad_por_frame"en cada fotograma- la posición vertical de la
bala*/
          from t = 1 to 60; //bucle del movimiento
                    x = x + v0x; //movimiento X
                    y = y + vy; //movimiento Y
                    vy = vy + gravedad_por_frame; //aceleración vertical de la velocidad
                    frame;
          end
END
```

Tal vez estés pensando que sería más lógico, en vez de utilizar un bucle FROM, utilizar un REPEAT/UNTIL (el proyectil volará HASTA que impacte con algo). Tienes razón. Para ello, deberíamos quitar el bloque FROM/END y poner en su lugar:

```
REPEAT
          x = x + v0x; //movimiento X
          y = y + vy; //movimiento Y
          vy = vy + gravedad_por_frame; //aceleración vertical
          frame;
UNTIL (...) /*fin del bucle. Aquí se escribirá la condición de ruptura:
              haber impactado con el escenario o con el enemigo.*/
```

El movimiento resultante, utilices el bucle que utilices, debería ser la trayectoria en forma de arco que apunta directamente al objetivo. Para hacer que el proyectil se elevara antes de llegar al objetivo, habría que calcular el movimiento teniendo en cuenta la altura máxima que se debe alcanzar.

Ojo. Hay que tener en cuenta que las coordenadas de pantalla no son iguales a las cartesianas de toda la vida, el eje y positivo es hacia abajo y empieza en la esquina superior izquierda de la pantalla por lo que la gravedad debe ser positiva, y la velocidad vertical inicial de un proyectil disparado con un angulo inicial positivo de menos de 180 grados debe ser negativa ya que va "hacia arriba".

Se recomienda usar muchos floats o aumentar la variable ***RESOLUTION*** para ganar precisión, porque si no a poco que los movimientos sean rápidos se vuelven muy bruscos. No obstante, es preferible la primera alternativa, porque utilizar ***RESOLUTION*** puede hacer que te despistes en algún cálculo y cueste encontrar el fallo.

En este tipo de juegos, el movimiento suele ser mucho más lento que en la realidad para que sea perceptible, eso implica una gravedad menor y una velocidad horizontal menor a la que adquiriría un proyectil real, para evitar una colisión que se produjera en décimas de segundo.

En la realidad también existe lo que se llama velocidad terminal, que es la máxima velocidad que puede alcanzar un cuerpo en caída libre a traves de un fluido, debido a que por el rozamiento de éste, la aceleración del rozamiento anula la de la gravedad. En un juego, salvo que sea de paracaidismo o algo por el estilo, solo hay que tener en cuenta esto por motivos esteticos más que por otra cosa. De hecho, en el código que acabamos de escribir, la velocidad puede crecer tanto que la trayectoria del misil no se vea. Para evitar esto bastaría un if:

```
if (vy > velocidad_terminal*resolution)
          vy = velocidad_terminal;
end
```

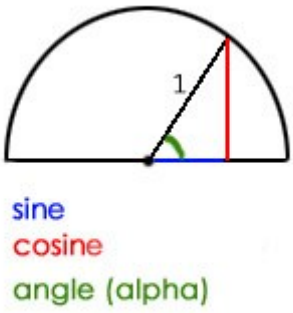
Una última aclaración: si queremos calcular el ángulo inicial, por el motivo que sea, a partir de este

movimiento, bastará con almacenar en variables los parámetros de X e Y para dos instantes distintos, correspondientes a la posición inicial y al segundo fotograma y echar mano de la trigonometría.

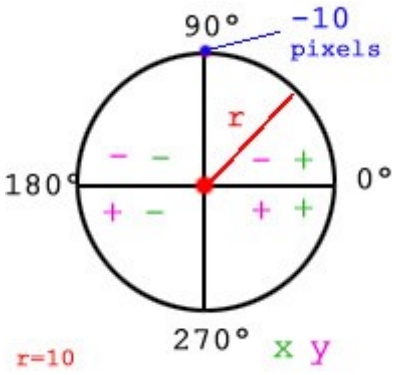
El ejemplo acabado de exponer aquí sobre el tiro parabólico es muy básico: se ha centrado en una parte de la física llamada cinemática, que es el estudio del movimiento. Pero también existe la llamada dinámica, que es el estudio de las fuerzas, y que nos puede dar muchísimo más jugo. Se sale completamente de nuestras posibilidades estudiar estos ámbitos, pero como muestra de los efectos realistas que se pueden llegar a conseguir utilizando las ecuaciones de la dinámica ($F=m \cdot a$,etc), aquí tienes un ejemplo de cuadrado que rebota en el suelo, con gravedad.

```
import "mod_video";
import "mod_map";
import "mod_key";
import "mod_proc";

Const
    anchura=640;
    altura=480;
End

Process main()
Private
          int x_0=anchura/2;
          int y_0=altura/4;
          int v_x=2;
          int v_y=0;
          int g=2;
//        int accel_x=1;
End
Begin
          set_mode(anchura,altura,32);
          graph = new_map(100,100,32);map_clear(0,graph,rgb(255,255,0));
          x=x_0; y=y_0; //Posición inicial
          LOOP
                    if((x+v_x >= anchura - map_info(0,graph,G_WIDTH)/2) || (x+v_x <=
map_info(0,graph,G_WIDTH)))
                              v_x = -v_x;
                    end
                    if(y+v_y >= altura - map_info(0,graph,G_HEIGHT)/2)
                              v_y = -v_y;
                    end
                    if(key(_esc))exit(); end

                    v_y =v_y+g;
                    x =x+v_x; //Actualizamos nuestra posición
                    y =y+v_y;
                    frame;
          End
End
```

Nos olvidaremos ahora del movimiento parabólico para centrarnos en otro tipo de trayectorias: a continuación veremos cómo generar trayectorias circulares. Antes de nada, para hacer trayectorias circulares necesitaremos utilizar trigonometría. Mira la imagen:




Aquí tenemos un semicírculo (de 0º a 180º). Si el radio del círculo es 1 –no importa si son metros o centímetros…-, el seno es la línea roja (en el lado opuesto al ángulo "alpha"), y el coseno es la línea azul (en el lado adyacente al ángulo "alpha"). Ahora imagina que queremos poner un enemigo en el punto del semicírculo donde llega el radio que tiene ese ángulo "alpha". Su X sería el coseno, y su Y el seno. Pero como el radio no siempre será 1, entonces haremos lo siguiente: *x = radio·coseno(alpha)* y *y = radio·coseno(alpha)* ¿OK? Ahora otro dibujo interesante:




Nota: Los + y – púrpuras son para la Y y los verdes para la X

Lo que tenemos aquí es un círculo de radio 10. Ahora imagina que queremos saber la coordenada Y cuando el ángulo (alpha) es 90º. El seno de 90º es 1. Como y=r·seno, obtenemos que y=10·1=10. Así que tenemos que situar nuestro enemigo 10 píxeles arriba. Pero ojo, la coordenada Y crece de arriba para abajo, así que la posición de este punto respecto el centro del círculo deberá ser de -10: las coordenadas Y entre 0º y 180º son negativas y entre 180º y 360º son positivas. Para las X, que crecen de izquierda a derecha, serían negativas entre 90º y 270º y positivas en la otra mitad.

Vamos a hacer un ejemplo. Supongamos que queremos que nuestro enemigo realice un movimiento circular constantemente (en plan amenazador, por ejemplo). Lo primero que hay que saber es cuántos fotogramas necesitará nuestro enemigo para completar una vuelta al círculo. Por ejemplo, si he elegido que tardará 36 frames en hacer una vuelta completa, entonces deberá avanzar 10º en cada frame (360º/36 frames=10º/frame).El siguiente paso es crear dos tablas, "*camino_x"* y "*camino_y"*, donde almacenaremos los resultados de calcular *y=r·seno* y *x=r·coseno*. Ya que estos resultados solamente habrá que calcularlos para una vuelta (son cíclicos), sería tontería irlos calculando una y otra vez a cada vuelta que hiciera el enemigo. Por eso es más inteligente calcularlos la primera vez y almacenarlos en un par de tablas, que serán las que a partir de entonces definirán el movimiento del enemigo.

Por cierto, la posición que guardaremos en las tablas se refiere al centro del círculo, no a la posición real del enemigo. Esto quiere decir que (en el proceso enemigo):

*x =x + camino_x[n];* <-- ¡INCORRECTO!

*y =y + camino_y[n];* <-- ¡INCORRECTO!
*x = centro_circulo_x + camino_x[n];* <--- ¡CORRECTO!
*y = centro_circulo_y + camino_y[n];* <--- ¡CORRECTO!

Así que crearemos dos tablas con 36 elementos cada uno, almacenaremos el resultado de hacer los cálculos de X e Y para los ángulos diferentes de 10º en 10º y un radio dado. Aquí está el código

```
import "mod_video";
import "mod_map";
import "mod_key";
import "mod_proc";

global
 int idenemigo;
 float camino_x[35];
 float camino_y[35];
 int i;
end

process main()
private
 float radio=100;
 float angulo=0;
end
Begin
    set_mode(320,240,32);
    //Cálculo de los elementos de las tablas que definirán el movimiento
    for(i=0;i<=35;i=i+1)
           camino_x[i]=radio*cos(angulo);
           camino_y[i]=radio*sin(angulo);
           angulo=angulo+10000;
    end

    //Se muestra en pantalla el enemigo
    idenemigo=new_map(30,30,32);map_clear(0,idenemigo,rgb(20,50,80));
    enemigo(160,120);

    Repeat
        Frame;
    Until (key(_esc))
    exit();
End

Process enemigo(x,y)
Private
  int centrox;
  int centroy;
End
Begin
    graph=idenemigo;
    centrox=x;
    centroy=y;
    Loop
        For(i=0;i<=35;i=i+1)
          //Las coordenadas hacen referencia al centro del círculo
          x=centrox+camino_x[i];
          y=centroy+camino_y[i];
          Frame;
        End
    End
End
```

A continuación presento un código, bastante similar al anterior, pero más flexible, ya que aparte de generar una trayectoria circular para un proceso, permite asociarle también **trayectorias elípticas**, donde el usuario (en este ejemplo concreto) puede variar además su radio mayor y menor, y resituar su punto central.

```
import "mod_video";
import "mod_map";
import "mod_key";
import "mod_proc";
import "mod_text";

process main()
begin
  set_mode(640,480,32);
  enemigo();
end

process enemigo()
private
          int centro_x=320;
          int centro_y=240;
          int radio_grande=50;
          int radio_peque=20;
          int angulo=0;
          int seno;
          int coseno;
end
begin
 graph=new_map(50,50,32);map_clear(0,graph,rgb(255,0,255));
 write(0,10,10,3,"flecha arriba--agranda en y");
 write(0,10,30,3,"flecha abajo--reduce en y");
 write(0,10,60,3,"flecha derecha--agranda en x");
 write(0,10,90,3,"flecha izquierda--reduce en x");
 write(0,10,320,3,"TECLADO NUMERICO MUEVE EL CENTRO");
 write(0,500,400,3,"ESC salir");
 write_var(0,10,230,4,radio_grande);
 write_var(0,10,250,4,radio_peque);
 write_var(0,320,20,4,centro_x);
 write_var(0,620,240,4,centro_y);
 loop

   //La clave del ejemplo está en estas cinco líneas
   seno=sin(angulo)*radio_peque;
   coseno=cos(angulo)*radio_grande;
   x=centro_x+coseno;
   y=centro_y+seno;
   angulo=angulo+15000;

   if(key(_right)and radio_grande<200)radio_grande=radio_grande+2;end
   if(key(_left)and radio_grande>20)radio_grande=radio_grande-2;end
   if(key(_up)and radio_peque<200)radio_peque=radio_peque+2;end
   if(key(_down)and radio_peque>20)radio_peque=radio_peque-2;end

   if(key(_8))centro_y=centro_y-2;end
   if(key(_2))centro_y=centro_y+2;end
   if(key(_4))centro_x=centro_x-2;end
   if(key(_6))centro_x=centro_x+2;end
   if(key(_esc))exit(0,"adios");end

   frame;
 end
end
```

Otro aspecto de la física clásica muy socorrido en el ámbito de los videojuegos es el de los **choques elásticos entre partículas** (por ejemplo, entre dos bolas de billar). Saber qué trayectorias, qué velocidad y aceleración tendrán determinados cuerpos tras haber colisionado entre sí puede ser de vital importancia para dotar al juego de un nivel de realismo imprescindible para captar el interés del usuario. Si se desea profundizar en el tema, recomiendo la lectura de los artículos http://www.sc.ehu.es/sbweb/fisica/solido/mov_general/choque/choque.htm y http://www.sc.ehu.es/sbweb/fisica/solido/percusion/percusiones.htm En general, el sitio http://www.sc.ehu.es/sbweb/fisica/default.htm es muy recomendable.

Finalmente, si se desea aprender más sobre la aplicación de la física a nuestros videojuegos, puedes

consultar la siguiente dirección: http://gpwiki.org/index.php/VB:Building_A_Simple_Physics_Engine_Part_1 Y más en concreto, si se desea obtener información muy completa sobre la física de las carreras de coches, puedes consultar http://www.racer.nl .Otra web interesante (sobre el sólido rígido y más cosas) es http://www.chrishecker.com/Rigid_Body_Dynamics

## *Uso de módulos no oficiales

Las posibilidades de Bennu no se acaban con los módulos que vienen "de serie" con él. Existe toda una multitud de módulos no oficiales que amplían las funcionalidades del lenguaje hasta límites insospechados, aportando nuevas órdenes extra, listas para realizar muy diversas tareas especializadas y potentes.

Para abarcar la amplitud de todos estos módulos de terceros, explicar las órdenes que incorporan, mostrar ejemplos de funcionamiento y uso, etc, se requeriría prácticamente un nuevo libro similar al presente. No disponemos de tanto espacio para ello, con lo que nos conformaremos en mostrar un breve listado de los módulos no oficiales más importantes ó interesantes, dejando al lector el trabajo de documentarse por su propia cuenta (cada módulo de hecho viene con un paquete de documentación y/o ejemplos, o bien se puede recurrir al foro de Bennu) y aprender por sí mismo el uso del módulo en el que esté interesado.

| | |
|---|---|
| **Bennu3D** (http://3dm8ee.blogspot.com) | Ambos módulos añaden a los juegos Bennu la posibilidad de utilizar modelos y animaciones 3D. Bennu3D está basado en el motor 3D Irrlicht (http://irrlicht.sourceforge.net) y también incorpora el motor de física Bullet (http://www.buletphysics.com) . Existen otras librerías que añaden funcionalidad 3D a Bennu, pero no están tan desarrolladas, o hace tiempo que han sido abandonadas, como "HiperGL" (http://forum.bennugd.org/index.php?topic=110.0) , la cual se base directamente en TinyGL (http://amitinygl.sourceforge.net , versión lite de OpenGL desactualizada) ó bien la librería "Sangine" ó "BeOgre". |
| **Fsock** (https://launchpad.net/~josebagar/+archive/ppa) **Net** (http://betatester.bennugd.org/snapshot/contrib) | Ambos módulos añaden a los juegos Bennu la posibilidad de comunicarse y transferir datos a través de una red TCP/IP -o también UDP- con otros ordenadores. La diferencia entre ellos dos radica en que mientras la Fsock es una librería programada desde cero, que accede directamente a los sockets a más bajo nivel (para crearlos, asociarlos a conexiones, ponerlos a la escucha, etc), la Net está basada en la librería SDL_Net (http://libsdl.org/projects/SDL_net) , por lo que libera al programador de cierto trabajo, pero añade una capa más de abstracción en el código. |
| **Mod_Sqlite** (https://launchpad.net/~josebagar/+archive/ppa) **OpenDBX** (http://forum.bennugd.org/index.php?topic=644.45) | Ambos módulos añaden a los juegos Bennu la posibilidad de acceder a bases de datos para escribir en ella datos que se deseen persistentes, para consultarlos, etc. La diferencia entre ellos dos radica en que mientras Mod_Sqlite está pensado tan sólo para interactuar con el servidor de base de datos Sqlite (http://www.sqlite.org) , el módulo de OpenDBX puede acceder a multitud de servidores de BBDD diferentes: Oracle, SQLServer, MySQL, PostgreSQL, etc, etc. Lógicamente, este último módulo es bastante más complejo y no tan liviano para distribuir. |
| **Mod_Image** (https://launchpad.net/~josebagar/+archive/ppa) | Módulo que se basa en la SDL_image (http://www.libdsl.org/projects/SDL_image) , el cual añade a los juegos Bennu la posibilidad de cargar multitud de formatos de imagen diferentes (JPG, GIF, TIFF, BMP, |

| | |
|---|---|
| | etc), no tan sólo PNG. |
| **Mod_Ttf** (https://launchpad.net/~josebagar/+archive/ppa) | Módulo que añade a los juegos Bennu la posibilidad de cargar y trabajar con fuentes de tipo TTF (True Type), transformándolas en memoria en el formato propio de Bennu. Tiene funcionalidad de anti-aliasing. Requiere de la librería FreeType (http://www.freetype.org) para funcionar |
| **Mod_Pango** (https://launchpad.net/~josebagar/+archive/ppa ) | Módulo que añade a los juegos Bennu la posibilidad de utilizar la funcionalidad de la librería Pango (http://www.pango.org) en el renderizado y dibujado de textos. |
| **Mod_Smpeg** (http://forum.bennugd.org/index.php?topic=866.0) **Mod_Mpeg** (https://launchpad.net/~josebagar/+archive/ppa) | Módulos que añaden ambos a los juegos Bennu la posibilidad de visualizar y controlar la reproducción de vídeos en formato Mpeg-1 en ellos, gracias al uso de la librería Smpeg (http://icculus.org/smpeg) |
| **FMODEx** (http://www.bennugd.org/downloads) | Módulo que añade a los juegos Bennu la posibilidad de utilizar la funcionalidad de la librería FMOD (http://www.fmod.org) para el uso del subsistema de audio, en vez de la librería estándar SDL_Mixer. La librería Fmod no eslibre, pero si el proyecto que la utiliza no es comercial, su uso es gratis. |
| **TinyXML** (http://forum.bennugd.org/index.php?topic=466.0) | Módulo que añade a los juegos Bennu la posibilidad de acceder al contenido de archivos XML, tanto para escribir en ellos datos que se deseen persistentes como para consultarlos, etc. Requiere la librería TinyXML (http://www.grinninglizard.com/tinyxml) |
| **SplinterGUI** (http://betatester.bennugd.org/snapshot/samples) | Esta librería no es un módulo propiamente dicho, sino más bien un conjunto de ficheros Prg (por tanto, código Bennu) que posibilitan la creación y manejo de elementos GUI estándar, tales como cuadros de diálogos, menús, botones, radios, checkboxs, etc. Permite pues poder programar con Bennu aplicaciones gráficas más estándar (tipo “oficina”). |

## *Programación (en lenguaje C) de módulos para Bennu

Lo primero que hay que indicar es que este apartado requiere unos conocimientos relativamente elevados del lenguaje C. Sin ellos no se podrá seguir correctamente las indicaciones que se darán a lo largo de él. Y ello es debido a que el propio Bennu (y, lógicamente, los módulos que lo componen) están programados en C (utilizando además la librería SDL, tal como ya sabemos). Este hecho es fácilmente comprobable si obtenemos el código fuente de Bennu bien descargándolo comprimido en un paquete tar.gz desde la web oficial, o bien descargándolo directamente del repositorio de Sourceforge. Por lo tanto, este apartado (y el siguiente) se desmarca un poco del resto de este manual, al estar enfocado a programadores con otro nivel de conocimientos.

Bennu ofrece una manera coherente y estable para aumentar su funcionalidad a través de módulos escritos por terceras personas. Debido a la estructura interna de Bennu, y a la correspondiente independencia entre los distintos módulos que lo componen, es muy fácil poder programar un módulo que realice la labor que deseamos en concreto sin necesidad de alterar para nada el comportamiento general del core, y ni tan sólo conocer cómo está programado éste internamente. Bennu está diseñado para que el programador que desea extender alguna funcionalidad concreta no tenga que saber ningún detalle interno y que simplemente utilizando una serie de sencillas reglas que explicaremos a continuación, pueda tener listo un módulo multiplataforma (Windows y GNU/Linux) a partir de una inversión mínima de esfuerzo y tiempo.

**Instalando y configurando el entorno**

Ya que para programar un módulo deberemos hacerlo con el lenguaje C, es evidente que lo más inmediato que necesitaremos será un compilador de C. Pero, no sólo eso: aunque no sea estrictamente imprescindible, también usaremos un IDE, ya que nos facilitará enormemente nuestro trabajo. El compilador de C más extendido y utilizado en el mundo con gran diferencia es el GNU Gcc, así que utilizaremos éste. Existen no obstante varios IDEs diseñados para programar en C, pero escogeremos el entorno CodeBlocks (http://www.codeblocks.org) , ya que es multiplataforma (funciona para Windows y para GNU/Linux igual), es libre y es un gran programa. Para instalar Gcc y Codeblocks en nuestro sistema, procederemos según los pasos marcados en la siguiente tabla:

| ***Windows*** | ***GNU/Linux*** |
|---|---|
| Debemos ir al apartado de descargas de la web de CodeBlocks (http://www.codeblocks.org).Allí tendremos 2 posibilidades: o bien descargarnos el IDE solamente, o bien descargarnos el IDE incluyendo "de serie" un compilador de C también. La primera opción nos sería útil si ya tuviéramos instalado previamente algún compilador de C, pero como en principio no será nuestro caso, elegiremos la segunda opción.<br><br>Fíjate que el compilador de C que viene incluido se llama MinGW y no Gcc; no te asustes: MinGW es una versión de Gcc para Windows (de hecho, la más importante y famosa). Por tanto, en el momento de escribir este texto,el paquete a descargar es *codeblocks-8.02mingw-setup.exe*.<br><br>Si por curiosidad, quisieras instalarte el compilador MinGW por separado independientemente de cualquier IDE, no es tan fácil como hacer clic en un instalador: deberías seguir los pasos que proponen en su página web oficial: http://www.mingw.org/wiki/HOWTO_Install_the_MinGW_GCC_Compiler_Suite | Todos los componentes necesarios se encuentran disponibles en los repositorios de cualquier distribución importante. Éstos son el entorno CodeBlocks propiamente dicho, el compilador Gcc propiamente dicho y la suite de compilación Make. Por ejemplo, en Ubuntu deberíamos hacer:<br><br>`aptitude install codeblocks gcc make` |

Además de un IDE y un compilador, necesitamos tener más cosas en nuestro ordenador para poder programar un módulo de Bennu. En concreto: el código fuente de Bennu, pero también los códigos fuentes de aquellas dependencias de Bennu que sean necesarias para la programación de nuestro módulo concreto. Es decir, los códigos fuentes de: la librería SDL, la librería SDL_Mixer , la librería LibPng y la librería Zlib (aunque dependiendo del caso no será necesario utilizar alguno de ellos). Para instalar todo ello en nuestro sistema, procederemos según los pasos marcados en la siguiente tabla:

| ***Windows*** | ***GNU/Linux*** |
|---|---|
| ***Código fuente Bennu:**<br>Ir a la página de descargas de la web oficial de Bennu, y descargar desde allí el "TAR source". Su URL en el momento de escribir este texto es http://bennugd.svn.sourceforge.net/viewvc/bennugd.tar.gz?view=tar Descomprimir su contenido en la carpeta que se desee; nosotros supondremos a partir de ahora que se ha descomprimido en *C:\devBennu\Bennu*<br><br>***Código fuente SDL:**<br>Ir a http://www.libsdl.org/download-1.2.php y descargar el paquete *SDL-devel-1.2.13-mingw32.tar.gz* (en el momento de escribir este texto...lo importante es que el nombre del paquete tenga el sufijo "devel"). Descomprimir su contenido en la carpeta que se desee; nosotros supondremos a partir de ahora que se ha descomprimido en *C:\devBennu\SDL* | ***Código fuente Bennu:**<br>Ir a la página de descargas de la web oficial de Bennu, y descargar desde allí el "TAR source". Su URL en el momento de escribir este texto es http://bennugd.svn.sourceforge.net/viewvc/bennugd.tar.gz?view=tar .Descomprimir su contenido en la carpeta que se desee; nosotros supondremos a partir de ahora que se ha descomprimido en */usr/include/Bennu*<br><br>***Código fuente SDL:**<br>Se encuentran disponibles en los repositorios de cualquier distribución importante. Por ejemplo, en Ubuntu deberíamos hacer (¡fijarse en el sufijo "dev"!, el paquete que queremos ha de llevarlo en su nombre, si no, no es el que necesitamos):<br><br>`aptitude install libsdl1.2-dev`<br><br>Este paquete normalmente tiene muchas dependencias, las |

| | |
|---|---|
| | cuales deberemos dejar que se instalen, automáticamente también.<br><br>Es importante, seguidamente de haber acabado el proceso de instalación automático de las cabeceras de la SDL, conocer dentro de qué ruta de carpetas el sistema ha copiado éstas, ya que este dato posteriormente lo utilizaremos. Para saber esta información, por ejemplo en Ubuntu podemos hacer:<br><br>`dpkg -L libsdl1.2-dev`<br><br>Normalmente, dicha ruta de instalación es */usr/include/SDL* |
| ***Código fuente SDL_Mixer:**<br>Ir http://www.libsdl.org/projects/SDL_mixer y descargar el paquete *SDL_mixer-devel-1.2.8-VC8.zip* (en el momento de escribir este texto...lo importante es que el nombre del paquete tenga el sufijo "devel"). Descomprimir su contenido en la carpeta que se desee; nosotros supondremos a partir de ahora que se ha descomprimido en *C:\devBennu\SDL_Mixer* | ***Código fuente SDL_Mixer:**<br>Se encuentran disponibles en los repositorios de cualquier distribución importante. Por ejemplo, en Ubuntu deberíamos hacer (¡fijarse en el sufijo "dev"!, el paquete que queremos ha de llevarlo en su nombre, si no, no es el que necesitamos):<br><br>`aptitude install libsdl-mixer1.2-dev`<br><br>Es importante, seguidamente de haber acabado el proceso de instalación automático de las cabeceras de la SDL_Mixer, conocer dentro de qué ruta de carpetas el sistema ha copiado éstas, ya que este dato posteriormente lo utilizaremos. Para saber esta información, por ejemplo en Ubuntu podemos hacer:<br><br>`dpkg -L libsdl-mixer1.2-dev`<br><br>Normalmente, dicha ruta de instalación es */usr/include/SDL* |
| ***Código fuente LibPng:**<br>Ir a http://www.libpng.org/pub/png/libpng.html y descargar el paquete (dentro del apartado "Source code") *libpng-1.2.40-no-config.tar.gz* .Descomprimir su contenido en la carpeta que se desee; nosotros supondremos a partir de ahora que se ha descomprimido en *C:\devBennu\LibPng* | ***Código fuente LibPng:**<br>Se encuentran disponibles en los repositorios de cualquier distribución importante. Por ejemplo, en Ubuntu deberíamos hacer (¡fijarse en el sufijo "dev"!, el paquete que queremos ha de llevarlo en su nombre, si no, no es el que necesitamos):<br><br>`aptitude install zlib1g-dev`<br><br>Es importante, seguidamente de haber acabado el proceso de instalación automático del código fuente de LibPng, conocer (tal como acabamos de hacer en párrafos anteriores) dentro de qué ruta de carpetas el sistema ha copiado éste, ya que este dato posteriormente lo utilizaremos. |
| ***Código fuente Zlib:**<br>Ir a http://www.zlib.org y descargar el paquete "source code" *zlib-1.2.3.tar.gz* (la zona de descargas está bastante abajo en esa página). Descomprimir su contenido en la carpeta que se desee; nosotros supondremos a partir de ahora que se ha descomprimido en *C:\devBennu\Zlib* | ***Código fuente Zlib:**<br>Se encuentran disponibles en los repositorios de cualquier distribución importante. Por ejemplo, en Ubuntu deberíamos hacer (¡fijarse en el sufijo "dev"!, el paquete que queremos ha de llevarlo en su nombre, si no, no es el que necesitamos):<br><br>`aptitude install libsdl-mixer1.2-dev`<br><br>Es importante, seguidamente de haber acabado el proceso de instalación automático del código fuente de Zlib, conocer (tal como acabamos de hacer en párrafos anteriores) dentro de qué ruta de carpetas el sistema ha copiado éste, ya que este dato posteriormente lo utilizaremos. |

Finalmente, también necesitaremos el entorno binario de Bennu instalado, ya que CodeBlocks lo necesitará.

Una vez ya instalados ó descargados todos estos paquetes que son un prerrequisito para poder trabajar, antes de empezar a trabajar, deberemos configurarlos convenientemente. Abramos el IDE CodeBlocks:

1.-Lo primero que tenemos que hacer es decirle a CodeBlocks dónde se encuentran los diferentes códigos fuentes (Bennu y dependencias) para que este entorno pueda localizarlos y utilizarlos cuando sea necesario. Esto se hace yendo al menú *"Settings->Compiler and debugger"* , y clicando en la pestaña *"Search directories->Compiler"* .Allí se deberán incluir las rutas de las cabeceras (archivos *.h) que vienen en los distintos códigos fuente a utilizar. En concreto, las cabeceras del código Bennu se encuentran (si hemos seguido nuestro ejemplo en Windows) en la subcarpeta ***C:\devBennu\Bennu\core\include*** y ***C:\devBennu\Bennu\core\bgdrtm\include*** . Si fuera necesario, además habría que añadir las cabeceras existentes dentro de las subcarpetas correspondientes a las librerías internas de Bennu, dentro de ***C:\devBennu\Bennu\modules***. También habría que incluir las rutas de las cabeceras de las distintas dependencias que se necesiten utilizar (SDL, SDL_Mixer, etc)

2.-También hemos de decirle a CodeBlocks dónde se ubica la librería **libbgdrtm.dll/libbgdrtm.so** que viene con el entorno binario de Bennu. Normalmente esta librería está dentro de la subcarpeta "bin". Así pues, deberemos indicar su ruta exacta (incluyendo su propio nombre) yendo al menú *"Settings->Compiler and debugger"* ,y clicando en la pestaña "Linker settings" , dentro del apartado *"Link libraries"* . Es posible que, dependiendo del módulo que se quiera programar, se necesiten incluir en esa lista más librerías, pero no suele ser habitual.

**Desarrollando una librería dinámica en C de ejemplo**

Lo más probable en el día a día es que nos encontremos con una librería dinámica ya desarrollada por otra persona, y lo único que queramos es desarrollar el módulo de Bennu correspondiente. O incluso que queramos desarrollar un módulo independiente que no utilice ninguna librería externa a él. En este caso, puedes saltarte este apartado, ya que aquí lo que haremos será desarrollar una librería propia muy muy sencilla que nos servirá de base para posteriormente acceder a ella a través del módulo Bennu que escribiremos en el apartado siguiente. Así que si ya tienes la librería en tus manos, pasa al punto siguiente. Eso sí, si no has programado tú esta librería de terceros, procura o bien tener acceso a su código fuente para conocer qué funciones ofrece para poder conectar a ellas desde tu módulo, o en su defecto, si no dispones del código fuente y sólo tienes la versión compilada (*.DLL para Windows, *.SO para GNU/Linux) procura tener acceso a la libre consulta de la documentación de ésta, donde deberá detallarse la API de la librería, de forma completa y actualizada.

Para crear una librería dinámica en C mediante CodeBlocks, lo único que debemos hacer es ir al cuadro de diálogo de "Nuevo Proyecto" (es decir, ir al menú *"File->"New project"*) y escoger de entre todas las posibilidades que nos ofrece, la opción "Shared library". En el cuadro siguiente se deberá escribir el nombre y la ubicación del proyecto; seguidamente nos dará la opción de elegir el tipo de compilador de C que queremos utilizar (nosotros siempre usaremos GNU GCC) y ya está. Los demás valores que nos muestra el asistente (diferentes rutas donde se almacenarán diversos archivos del proyecto) los dejaremos con sus valores por defecto. No obstante, existe un detalle muy importante que no se nos debe pasar en este punto: la elección del nombre del proyecto. El nombre del proyecto determina el nombre de la librería que crearemos, ya que éste será automáticamente el del proyecto precedido de "lib". Es decir, si nuestro proyecto se llama "pepito", la librería finalmente generada se llamará "libpepito.dll" (ó "libpepito.so"). Esto es importante tenerlo en cuenta.

Una vez finalizado el asistente, CodeBlocks nos generará automáticamente un esqueleto de código llamado "main.c" con una serie de funciones de muestra. Podríamos utilizar este código de ejemplo, o bien programarnos nosotros las funciones C que deseáramos, o incluso copiar/pegar el código fuente de una librería externa y compilarla. De cualquier manera, acabaríamos finalmente teniendo el archivo *.dll ó *.so sobre el cual podríamos montar nuestro módulo Bennu. Para ello, para generar dicha librería final, simplemente deberemos darle al botón de compilar: en Windows se obtendrá un archivo DLL y en GNU/Linux un archivo SO (dentro de la subcarpeta "Debug" de nuestro proyecto). Este archivo será lo único que necesitemos para realizar el paso siguiente: desarrollar un módulo Bennu que pueda acceder a las funciones que acabamos de escribir en "main.c" y que están incluidas dentro de dicho archivo compilado.

Como ejemplo a desarrollar, haremos una librería que constará de las siguientes funciones:

*Una función "Fibonacci()" -recursiva-, que recibe el número de elementos que se desea que tenga la

sucesión de Fibonacci a estudiar, y que devolverá la suma total de esos elementos.

*Una función “Factorial()” -recursiva- que recibe un parámetro entero y devuelve su factorial

*Una función “mirrorStr()”, que recibe como parámetro una cadena y la devuelve “del revés”.

He aquí el código (el fichero compilado lo llamaremos “libdctools.dll/.so”):

```
#include <string.h>  //Necesaria por el uso de la función estándar strlen()

int fibonacci(int n) {
          int ret = 1;
          if ((n != 0) && (n != 1))
                    ret = fibonacci(n - 1) + fibonacci(n - 2);
          return ret;
}

unsigned int factorial(unsigned int n) {
          unsigned int ret = 1;
          if (n > 1)
                    ret = n * factorial(n - 1);
          return ret;
}

char * mirrorStr(char * str) {
          int i, len;
          char aux;
          len =strlen(str);
          for (i = 0;i<len/2; i++) {
                    aux=str[len-1-i];
                    str[len-1-i]=str[i];
                    str[i]=aux;
          }
          return str;
}
```

**Desarrollando un módulo de Bennu que utiliza la librería dinámica del apartado anterior**

Crear un módulo de Bennu es muy similar a crear una librería dinámica en C. De hecho, es lo mismo, sólo que hay que tener algún detalle más en cuenta. Primero comentaremos la estructura general del código que ha de tener cualquier módulo Bennu, y posteriormente aplicaremos este conocimiento al desarrollo del módulo que pueda hacer uso de la librería generada en el apartado anterior, y por tanto, de sus tres funciones definidas en ella: “fibonacci()”, “factorial()” y “mirrorStr()”.

Para empezar, en cualquier módulo de Bennu (como código C que es) se necesitarán incluir determinadas cabeceras, que pueden variar, evidentemente, según las funciones que se vayan a usar, pero que siempre serán como mínimo dos:

```
#include "bgddl.h"
#include "bgdrtm.h"
```

Seguidamente, como en cualquier otro código de C, se podrán declarar las variables internas que se requieran para nuestro módulo, se puede incorporar alguna línea #define, etc. Y a partir de aquí, ya depende.

*Para definir constantes que se puedan utilizar desde Bennu, deberíamos escribir:

```
DLCONSTANT  __bgdexport( nombremodulo, constants_def )[] =
{
   { "NOMBRE_CONSTANTE", TYPE_INT, 0  },
   { "NOMBRE_CONSTANTE", TYPE_INT, 1  },
   { NULL          , 0       , 0  }
} ;
```

donde “nombremodulo” (y esto es muy importante) deberá obligatoriamente ser del tipo “mod_aaa” donde *aaa* es el nombre de la librería *.dll/*.so en la que se basa el módulo, (pero sin el prefijo “lib”). Si el módulo no se basara en ninguna librería externa, sino que fuera independiente, el nombre del módulo también sería “mod_aaa” pero entonces

*aaa* puede ser cualquier cosa. Esta regla se deberá cumplir siempre que aparezca "nombremodulo" en los siguientes trozos de código.

Por otro lado, se puede comprobar que lo que se está realmente haciendo es definir una matriz especial que consta de elementos (las constantes, propiamente) que se componen a su vez de tres elementos (el nombre de la constante, el tipo de la constante y el valor de la constante). Los tipos de las constantes pueden ser TYPE_INT, TYPE_DWORD, TYPE_SHORT, TYPE_WORD, TYPE_BYTE, TYPE_FLOAT ó TYPE_STRING. Es importante respetar las mayúsculas, tanto especificando el tipo de la constante como también el nombre de las constantes (entre comillas). E Importante tener en cuenta también que siempre hay que incluir un elemento NULL (sin comillas) de tipo 0 y valor 0, obligatoriamente.

*Para definir variables globales que se puedan utilizar desde Bennu:

```
char * __bgdexport( nombremodulo, globals_def ) =
   "STRUCT una_estructura\n"
   " un_campo;\n"
   " otro_campo;\n"
   " otro_mas;\n"
   "END \n"
   "int una_variable;\n" ;

DLVARFIXUP  __bgdexport( nombremodulo, globals_fixup )[] =
{
   { "una_estructura.un_campo", NULL, -1, -1 },
   { "una_estructura.otro_campo", NULL, -1, -1 },
   { "una_estructura.otro_mas", NULL, -1, -1 },
   { "una_variable", NULL, -1, -1 },
   { NULL, NULL, -1, -1 }
};
```

Aquí se puede ver que las variables globales las debemos declarar primero con sintaxis Bennu, pero siempre rodeando cada línea entre comillas, y acabándolas con "\n", tal como queramos que sea utilizada y reconocida por el código Bennu que las requieran. Y después, debemos definir una matriz especial que consta de elementos (las variables globales, propiamente -o en el caso de estructuras, cada campo individual-) que se componen a su vez de cuatro elementos (el nombre de la variable, el puntero al dato, el tamaño del elemento y la cantidad de elementos). De éstos, el único elemento que deberemos tener en cuenta será el primero: el resto podrán valer siempre respectivamente NULL, -1 y -1, sin tener que preocuparnos por nada más. Importante tener en cuenta también que siempre hay que incluir un último elemento con los valores NULL, NULL, -1 y -1, obligatoriamente.

*Para definir variables locales que se puedan utilizar desde Bennu:

```
char * __bgdexport ( nombremodulo,locals_def ) =
       "int loc1;\n"
       "int loc2;\n" ;

DLVARFIXUP  __bgdexport ( nombremodulo,locals_fixup ) []  =
{
     { "loc1", -1, -1, -1 },
     { "loc2", -1, -1, -1 },
     { NULL, -1, -1, -1 }
};
```

Aquí se puede ver que las variables locales las debemos declarar primero con sintaxis Bennu, pero siempre rodeando cada línea entre comillas, y acabándolas con "\n", tal como queramos que sea utilizada y reconocida por el código Bennu que las requieran. Y después, debemos definir una matriz especial que consta de elementos (las variables locales, propiamente -o en el caso de estructuras, cada campo individual-) que se componen a su vez de cuatro elementos (el nombre de la variable, el puntero al dato, el tamaño del elemento y la cantidad de elementos). De éstos, el único elemento que deberemos tener en cuenta será el primero: el resto podrán valer siempre respectivamente -1, -1 y -1, sin tener que preocuparnos por nada más. Importante tener en cuenta también que siempre hay que incluir un último elemento con los valores NULL, -1, -1 y -1, obligatoriamente.

*Para definir tipos de datos personalizados (TYPEs) que se puedan utilizar desde Bennu:

```
char * __bgdexport( nombremodulo, types_def ) =
   "TYPE tipoDatos\n"
   " int uncampo;\n"
   " int otrocampo;\n"
   " int otromas;\n"
   " pointer pointer yotro;\n"
   "END\n"  ;
```

Aquí se puede ver que los tipos de datos personalizados los debemos declarar con sintaxis Bennu, pero siempre rodeando cada línea entre comillas, y acabándolas con "\n", tal como queramos que sea utilizada y reconocida por el código Bennu que las requieran.

*Para definir las funciones que estarán disponibles para Bennu en este módulo:

```
static int modnombre_funcion(INSTANCE * my, int * params) {
     … //Código de la función
}

static int modnombre_otrafunc(INSTANCE * my, int * params) {
     … //Código de la función
}


DLSYSFUNCS  __bgdexport( nombremodulo, functions_exports )[] =
{
   { "NOMBRE_FUNCION" , ""  , TYPE_INT     , modnombre_funcion },
   { "NOMBRE_OTRAFUNC", "I" , TYPE_STRING  , modnombre_otrafunc},
   { 0             , 0       , 0           , 0                 }
};
```

Lo primero que hay que tener en cuenta es que los nombres de las funciones que definiremos dentro del código C siempre vendrán precedidos del prefijo "mod". Esto lo haremos para distinguirlos de los nombres que tendrán esas mismas funciones en el código Bennu, que serán los nombres que el programador de Bennu hará servir cuando utilice nuestro módulo.

Otro aspecto es que TODAS las funciones que se definan dentro del código C tendrán el mismo prototipo. Es decir, tal como se puede observar en el cuadro anterior, deberán ser estáticas, devolver un valor de tipo INT y sus parámetros serán siempre dos: `(INSTANCE * my,int * params)` donde "params" funciona como un vector cuyos elementos son los distintos valores que se le pasarán a los parámetros que tenga la función Bennu (params[0] contendrá el valor del primer parámetro de la función Bennu, params[1] el segundo, etc).

Finalmente, se puede ver que también debemos definir una matriz especial que consta de elementos (las funciones, propiamente) que se componen a su vez de cuatro elementos (el nombre de la función que se utilizará en código Bennu -IMPORTANTE: SE HA DE ESCRIBIR EN MAYÚSCULAS SIEMPRE, y entre comillas-, el tipo y número de parámetros que tendrá dicha función Bennu -se comentará en el siguiente párrafo-, el tipo de datos que devolverá dicha función -los tipos posibles se han indicado unos párrafos más arriba-, y el nombre de la función definida en el código C del propio módulo, que se corresponda con la función Bennu a exportar -recordemos que en su nombre deberá incorporar el prefijo "mod"-). Importante tener en cuenta también que siempre hay que incluir un último elemento con los valores 0,0,0 y 0, obligatoriamente.

Respecto el tipo y número de parámetros (el segundo valor de cada elemento dentro del vector de funciones), se sigue la siguiente nomenclatura: si la función Bennu no tuviera parámetros, se escribiría "". Si tuviera un parámetro de tipo entero, se escribiría "I". Igualmente, si el parámetro fuera decimal, sería "F". Si fuera de tipo cadena, sería "S". Y si fuera un puntero, sería "P". Si la función tuviera dos parámetros de tipo entero, se escribiría "II". O si tuviera tres: una cadena, un decimal y un entero (por este orden), se escribiría "SFI". Es fácil intuir la lógica.

Por otro lado, otras cosas más exóticas:

*Para ejecutar un determinado código en el mismo momento que el módulo se cargue dentro de código Bennu:

```
void  __bgdexport( nombremodulo, module_initialize )()
{
```

```
  … //Sentencias
}
```

*Para ejecutar un determinado código en el mismo momento que el módulo se descargue dentro de código Bennu:

```
void  __bgdexport( nombremodulo, module_finalize )()
{
  … //Sentencias
}
```

*Para ejecutar un determinado código una vez cada frame. En cada pareja-elemento de la matriz, el primer valor (un número) representa el biggest priority first execute y el segundo valor (el nombre de una función, por ejemplo) representa el lowest priority last execute:

```
HOOK __bgdexport (libwm , handler_hooks ) []  =
{
   (  4700 ,  cosas ),
   (  0 , null ),
}
```

*Para declarar dependencias del módulo actual respecto alguna de las librerías internas de Bennu:

```
char * __bgdexport( nombremodulo, modules_dependency )[] =
{
    "libsdlhandler",
    "libgrbase",
    "libvideo",
    "libblit",
    NULL
};
```

Bien. Una vez ya conocida la estructura básica de un módulo genérico (aunque se puede aprender mucho más sobre esto si se consulta directamente el código fuente de un módulo cualquiera), vamos a generar un módulo que haga uso de la librería que compilamos en el apartado anterior. Para ello, deberemos volver a empezar un proyecto nuevo mediante el asistente, y volveremos a escoger como tipo de nuevo proyecto la opción "Shared library". Rellenaremos las pantallas del asistente tal como ya sabemos, y finalmente volveremos a tener generada la plantilla "main.c".

Es muy importante también que añadamos la librería que recién acabamos de compilar en el punto anterior a la lista de librerías que se van a utilizar para generar nuestro módulo, porque si no, cuando CodeBlocks quiera saber qué significa "fibonacci()" ó "factorial()", sepa en qué lugar fueron definidas. Por lo que deberemos de volver a ir al menú *"Settings->Compiler and debugger"* ,clicar en la pestaña "Linker settings" ,y añadir nuestra librería a vincular, dentro del apartado *"Link libraries"* Es todo lo que tenemos que hacer.

A continuación se presenta el código de nuestro módulo (que una vez compilado lo llamaremos "mod_dctools.dll/.so"), donde se puede apreciar que no hay ni mucho menos todas las secciones descritas arriba: tan sólo las necesarias (es decir, básicamente la definición de las tres funciones de la librería). Además, para completar un poco más nuestro módulo, se han añadido dos funciones, "ln()" y "log()" que no hacen uso de ninguna librería externa (solamente la librería estándar math.h de C) pero que incorporándolas a nuestro módulo podremos hacer que Bennu también tenga funciones logarítmicas, cosa que ya vimos  que Bennu no ofrecía por sí mismo pero que C sí que ofrece de forma estándar. Así pues, mediante nuestro módulo  posibilitaremos además que la potencia de cálculo de logaritmos de la que C dispone, pueda ser también utilizada por Bennu.

```
#include "bgddl.h"
#include <stdio.h>
#include <string.h>
#include <math.h>
//Necesaria para el uso de funciones de cadena (string_get(), string_new(), etc)
#include "xstrings.h"
```

```
static int moddctools_dclog(INSTANCE * my, int * params){
    //Se ha realizar un cast de int a float!!
    float num = *( float * ) &params[0];
    float result;
    result=log(num);
/*Aquí también hay un cast para que el valor retornado sea float. Consultar el código de
mod_math.c para más información */
    return *(( int * )&result );
}

static int moddctools_dclog10(INSTANCE * my, int * params){
    float num = *( float * ) &params[0] ;
    float result;
    result=log10(num);
    return *(( int * )&result );
}

static int moddctools_dcfibonacci(INSTANCE * my, int * params) {
    const int value = params[0];
    int result;
    result = fibonacci(value);
    return result;
}

static int moddctools_dcfactorial(INSTANCE * my, int * params) {
    const unsigned int value = params[0];
    unsigned int result;
    result = factorial(value);
    return result;
}

static int moddctools_dcmirrorstr(INSTANCE * my, int * params) {
          //Se usa la función string_get para obtener parámetros de tipo string.
           char * str = string_get(params[0]);
           printf("%s\n", str);
           char * result;
/*Funciones de xstrings.h que se puede utilizar: string_new, string_u/lcase, string_use,
string_substr, etc, etc. Consultar el código de mod_string.c para más información*/
           result = string_new(mirrorStr(str));
           //Siempre, antes del return, se ha de usar la función string_discard.
           string_discard(params[0]);
           return result;
}


DLSYSFUNCS __bgdexport( mod_dctools, functions_exports) [] = {
        {"DCLOG"      , "F", TYPE_FLOAT , moddctools_dclog},
        {"DCLOG10"    , "F", TYPE_FLOAT , moddctools_dclog10},
        {"DCFIBONACCI", "I", TYPE_INT   , moddctools_dcfibonacci},
        {"DCFACTORIAL", "I", TYPE_INT   , moddctools_dcfactorial},
        {"DCMIRRORSTR", "S", TYPE_STRING, moddctools_dcmirrorstr},
        {0, 0, 0, 0}
};
```

Hay que decir que, en el ejemplo que hemos hecho, no habría sido necesario separar las definiciones de las funciones “fibonacci()”, “factorial()” y “mirrorStr()” en una librería separada del módulo. Se podría haber incluido perfectamente su código dentro del código del módulo, incorporando en un sólo fichero fuente todo lo necesario, y por tanto, teniendo sólo un fichero *.dll/*.so compilado resultante. Pero se ha optado por explicarlo de forma separada porque es el caso más habitual.

Una vez ya compilado nuestro módulo (y obtenido por tanto, nuestro fichero *.dll/*.so correspondiente), en GNU/Linux hay que tener una precaución extra que es copiar el nuevo archivo .so generado a la carpeta de los módulos de Bennu (*/usr/lib/bgd/module*) y, por si acaso, incluir esta ruta en la variable de entorno *LD_LIBRARY_PATH* .Además de asegurarse de que dicho fichero .so tenga permisos de ejecución

Y un código Bennu de ejemplo de uso del módulo acabado de generar:

```
import "mod_key";
import "mod_text";
import "mod_dctools";

Process Main()
Private
          int resulti[2];
        float resultf[2];
          string resultc;
End
Begin
    resulti[0]=dcfibonacci(8);
    resulti[1]=dcfactorial(4);
    resultf[0]=dclog(4.6);
    resultf[1]=dclog10(5.93);
    resultc = dcmirrorStr("hola mundo");

    write_var(0,0,10,0,resulti[0]);
    write_var(0,0,20,0,resulti[1]);
    write_var(0,0,30,0,resultf[0]);
    write_var(0,0,40,0,resultf[1]);
    write_var(0,0,50,0,resultc);

    while (!key(_esc))
        frame;
    end
end
```

## *Utilizar Bennu como lenguaje de script embebido en código C

Si eres programador habitual en lenguaje C, es posible que en alguna ocasión hayas echado de menos la posibilidad de incluir “trozos” compilados de código Bennu dentro de tu código principal escrito en C, para que realizaran tareas concretas, pudiéndolas codificar pues de una forma más fácil y rápida que si se tuvieran que programar directamente en C (que, dicho sea de paso, un lenguaje bastante más complejo y delicado de usar). Bien, pues se puede hacer: podemos “embeber” código compilado Bennu dentro de nuestro código C. Es decir, podemos hacer que desde un programa escrito en C se pueda cargar un DCB y llamar a funciones escritas en lenguaje Bennu contenidas dentro de él. Este método también se podría utilizar para llamar desde un módulo compilado Bennu (fichero .dll en Windows, .so en GNU/Linux) a funciones de un dcb.

En general, un lenguaje de “scripting” es un tipo de lenguaje de programación que es generalmente interpretado. Por lo general, los scripts permanecen en su forma original (su código fuente en forma de texto) y son interpretados comando por comando cada vez que se ejecutan. Los scripts suelen escribirse más fácilmente, pero con un costo sobre su ejecución. En general, los scripts suelen ser almacenados como texto sin formato y suelen ser más pequeños que el equivalente en un lenguaje de programación compilado. Un “scripting embebido” es un scripting que se puede embeber en otro lenguaje:esto significa que podemos, desde un ejecutable X (compilado y cerrado), llamar a un scripting (escrito en Bennu, por ejemplo) para resolver cierta lógica y obtener el resultado de ésta. Esto nos permite poder tener un core binario estandard, y tener las particularidades custom/especiales de nuestro desarrollo en el scripting, lo cual tambien nos facilita el ajustar (nosotros o incluso alguien que no sabe programar) cosas o modificar la lógica sin tener que recompilar nuestro core o programa principal (binario cerrado), solo compilando el scripting, que será llamado otra vez por el programa principal.

De hecho, muchos juegos comerciales (a pesar de estar escritos en C o algun otro poderoso lenguaje) usan lenguajes de scripting embebidos para hacer ciertas cosas, o para tener partes de la lógica fuera del codigo compilado (binario del juego), y poder de esta forma tocar y ajustar cosas de forma muy fácil. Ejemplo de esto son muchos juegos de Lucas Arts y Lua... o incluso Quake y su “quakec”, por nombrar algunos.

Veamos cómo se hace todo esto con un ejemplo. Partamos del código Bennu que deseamos incluir en nuestro programa. En nuestro caso sería éste:

```
import "mod_rand";
```

```
import "mod_file";
import "mod_string";
import "mod_say";


process add(int a, int b)
begin
    say( a + "+" + b + "=" + (a+b));
    return (a+b);
end

process create_file(string f)
private
    int n;
    int fp;
    int v1,v2;
end
begin
    say("create_file entry");
    fp = fopen(f, O_WRITE);
    for (n = 0; n < 10; n++)
        v1 = rand(1,100);
        v2 = rand(1,100);
        fputs(fp,"add("+v1+","+v2+")="+add(v1,v2));
    end
    fclose(fp);
    say("create_file exit");
end

process main()
begin
end
```

Como se puede ver, el código anterior es simplemente un listado -previa importación de los módulos correspondientes que sean necesarios- de diferentes procesos (también pueden ser funciones) que ofreceremos a nuestro programa en C. En concreto, tenemos un proceso que recibe dos parámetros y devuelve su suma y otro que crea un fichero de texto y escribe en él la suma (utilizando el otro proceso acabado de definir) de dos números aleatorios, además de mostrar diferente información por consola. Atención, es importante también escribir el proceso "Main()" aunque no realice ninguna acción (en estos casos siempre suelen estar vacíos de código) porque si no al intentar compilar, *bgdc* nos devolverá un error de que no encuentra dicho proceso principal. Bien, pues ya está: sólo tendremos que compilar este código y listo. El nombre que le daremos al fichero dcb generado será por ejemplo "sample.dcb".

Ahora lo que falta es saber cómo podemos llamar desde nuestro código C a los dos procesos creados en el código anterior. Si somos programadores habituales de C, tendremos un entorno de desarrollo+compilador que será nuestro favorito. En este manual recomendamos CodeBlocks (http://www.codeblocks.org) ya que es muy buen programa, es libre y multiplataforma,

Lo primero que deberemos hacer es preparar nuestro entorno para que pueda encontrar las librerías internas de Bennu (en concreto, sus cabeceras *.h), ya que el código C que compilemos las necesitará. Para ello, habrá que tener el código fuente de Bennu descargado previamente y acceder al menú del CodeBlocks llamado "Settings"->"Compiler and debugger". En la ventana que aparece , deberemos clicar en la pestaña "Search directories", y dentro de ahí ir a la pestaña "Compiler", donde deberemos añadir las carpetas del código fuente de Bennu que contienen los ficheros que nuestro programa en C necesitará para compilarse correctamente. En concreto, suponiendo que tenemos Windows y hemos descargado el código fuente de Bennu en la carpeta C:\CodBennu ,estas carpetas son:

```
* C:\CodBennu\bennugd\core\include
* C:\CodBennu\bennugd\core\bgdi\include
* C:\CodBennu\bennugd\core\bgdrtm\include
```

Dependiendo del código Bennu que hayamos programado, éste requerirá el uso de alguna de las dependencias que Bennu tiene, por lo que en ese caso es necesario indicarle a CodeBlocks dónde encontrar las cabeceras de estas librerías externas, incluyendo las rutas a los códigos fuentes de la SDL, la SDL_Mixer, la LibPNG ó la Zlib, dependiendo del caso. Por ejemplo, en el código Bennu que se acaba de presentar antes, se hace uso de la

librería Zlib (en concreto, el módulo “mod_file” hace uso de ésta), por lo que además de las tres rutas anteriores, habría que añadir en el mismo lugar la ruta del código fuente de la Zlib, previamente descargado. Por ejemplo:

```
*C:\CodBennu\zlib-1.2.3
```

Si no supieras qué librerías externas necesitaras incluir en un determinado momento, puedes observar los errores de compilación que te muestre CodeBlocks y deducir a partir de allí qué ficheros *.h no se encuentran y han de ser encontrados.

Otra cosa que también hemos de decirle a CodeBlocks es dónde se ubica la librería **libbgdrtm.dll/libbgdrtm.so** que viene con el entorno binario de Bennu. Normalmente esta librería está dentro de la subcarpeta “bin”. Así pues, deberemos indicar su ruta exacta (incluyendo su propio nombre) yendo al menú *“Settings->Compiler and debugger”* ,y clicando en la pestaña "Linker settings" , dentro del apartado *"Link libraries"* . Es posible que, dependiendo del módulo que se quiera programar, se necesiten incluir en esa lista más librerías, pero no suele ser habitual.

Si no usaras el entorno Codeblocks, cualquier otro entorno de programación de C tendrá en sus menús de configuración una opción similar a la que hemos usado en CodeBlocks: hay que buscar dónde se tienen que especificar las rutas de búsqueda de librerías *.h para el compilador.

Una vez ya configurado el entorno, podemos proceder a crear un nuevo proyecto (menú “Nuevo->Proyecto” y seguir los pasos que marca el asistente (por ejemplo, nos preguntará qué tipo de compilador vamos a usar: de C ó de C++; nosotros elegiremos el primero). El código de ejemplo que mostraremos a continuación será de un programa de consola, por lo que en el asistente deberíamos seleccionar en este caso el tipo de proyecto “Console Application”.

Una vez finalizado el asistente, podremos proceder a escribir el código fuente de nuestro programa en C. A continuación presento un ejemplo comentado

```
#ifdef _WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "bgdi.h" //Casi obligatoria incluir siempre
#include "bgdrtm.h" //Casi obligatoria incluir siempre
#include "offsets.h" //Casi obligatoria incluir siempre
//Necesaria para el uso de funciones de cadena (string_init,string_new,string_use, etc)
#include "xstrings.h"

int main (int argc, char **argv)
{
/*Estas tres variables (r, proc y ret) siempre deberán ser escritas al principio de
nuestro código C */
    INSTANCE * r ;
    PROCDEF * proc;
    int ret;
/*Inicialización. Se han de incluir siempre estas dos funciones (como mínimo) al
principio de todo,ya que inicializan los módulos módulos necesarios antes del ejecutar la
función dcb_load()*/
    string_init() ;
    init_c_type() ;
    //Se carga el archivo dcb
    if (!dcb_load("sample.dcb")) {
        printf ("Error loading dcb\n") ;
        return -1;
    }
/Inicialización.Se ha de incluir siempre esta función (como mínimo) después de dcb_load()
    sysproc_init () ;
```

```
/* ---------- Uso de la función ADD definida en sample.dcb ---------------------- */
//Se obtiene la definición de la función (¡su nombre ha de estar en mayúsculas!)
    proc = procdef_get_by_name ("ADD");
    if (!proc)
    {
        printf ("Procc 'ADD' not found\n") ;
        return -1 ;
    }
    r = instance_new (proc, 0) ; //Se crea la función para ser usada en código C
/*PRIDWORD es una macro definida en instance_st.h. Lo único que nos interesará saber es
que su primer argumento representa la función "creada" por instance_new a la que queremos
asignar los valores de sus parámetros, y su segundo argumento es el número de byte
(empezando por 0) donde estará colocado en memoria cada uno de los valores a asignar como
parámetros a la función. Es decir: el primer parámetro siempre estará en el byte número
0, el segundo parámetro estará después del primero, a una cantidad de bytes determinada
según el tipo de parámetro previo (si es un Int, estará a 4 bytes del primero, si es de
tipo float también, si es de tipo word, será a 2 bytes, etc) ocupa cada valor*/
    PRIDWORD(r, 0) = 12 ;       //Se asigna el valor al primer parámetro
    PRIDWORD(r, 4) = 32 ;       //Se asigna el valor al segundo parámetro
    ret = instance_go(r) ;      //Se ejecuta la función y se obtiene el valor devuelto
    printf ("%d\n", ret);

/* ---------- Uso de la función CREATE_FILE definida en sample.dcb --------------- */
    proc = procdef_get_by_name ("CREATE_FILE");
    if (!proc)
    {
        printf ("Procc 'CREATE_FILE' not found\n") ;
        return -1 ;
    }
    r = instance_new (proc, 0) ;
//Con cadenas (el primer y único parámetro) se ha de utilizar las funciones de xstrings.h
    PRIDWORD(r, 0) = string_new("file.dat") ;
    string_use(PRIDWORD(r, 0));
    ret = instance_go(r) ;
    printf ("%d\n", ret);


/*Se acaba el programa.
También existe la función contraria: bgdrtm_entry(int argc, char * argv[])*/
//Los valores de su parámetro pueden ser los mismos que la función exit() estándar de C
    bgdrtm_exit(0);
    return 0;
}
```

Si todo va normal, deberemos obtener nuestro ejecutable sin problemas. A partir de aquí, cuando queramos distribuir nuestro programa compilado, deberemos tener en cuenta también de acompañarlo (lógicamente) del fichero dcb de soporte y de todas aquellas librerías (tanto módulos de Bennu como librerías externas como la Zlib o similares) que dicho dcb necesite, tal como si estuviéramos distribuyendo un videojuego.

## *¿Y ahora qué? Recursos para seguir aprendiendo

Ya hemos casi finalizado este manual. Tan sólo queda el último capítulo, que consiste en una miscelánea de códigos y trucos. Si has ido siguiendo todas las indicaciones, explicaciones y ejemplos dados capítulo a capítulo, en estos momentos te puedes considerar ya un buen programador de Bennu. No obstante, el camino no se acaba aquí: siempre hay aspectos del lenguaje o de la programación de videojuegos en general que se pueden aprender o mejorar. Nuestra actitud como programadores ha de ser siempre la de estar abiertos a nuevos retos, a nuevas aventuras; tener una curiosidad innata por descubrir nuevas maneras de hacer las cosas, o directamente, maneras de hacer nuevas cosas.

Además, en los apartados preliminares de este texto ya se indica que (como no puede ser de otra manera), en este manual no se tratan muchos temas que por falta de espacio o de cohesión con el resto del libro no se han incluido. Todo lo referente a los módulos de terceros (Fsock, Bennu3D, etc) no se explican, ni tampoco se profundiza en temas avanzados como por ejemplo el manejo de memoria dinámica (sus utilidades son infinitas). Por ello,

recomiendo fervientemente conseguir otras fuentes de información para ampliar conocimientos y permanecer así siempre a la última y siempre abarcando nuevos horizontes.

Algunos recursos que puedes utilizar para seguir aprendiendo Bennu son (todos ya los hemos nombrado en algún momento de este manual):

***BennuPack** (http://bennupack.blogspot.com) : Incluye muchísimos códigos de ejemplo clasificados por dificultad (básicos, medios, avanzados), siendo una fuente de información de un valor precioso. También incluye multitud de módulos externos para muy diversos propósitos (muchos de los cuales ni se han nombrado en este texto), con su respectiva documentación para ser utilizados

***Apagame4be** (http://code.google.com/p/apagame4be): Conjunto de juegos clásicos especialmente pensado precisamente para poder estudiar el código fuente y aprender a partir de ejemplos prácticos. Se ha primado la claridad de los códigos antes que la funcionalidad ó complejidad de éstos. Tenemos juegos como el Tetris, Conecta4, Galaxian, Pong, etc

***Foro oficial de Bennu** (http://forum.bennugd.org) y **Wiki** (http://wiki.bennugd.org) : Cómo no, el recurso más importante que tiene el programador de Bennu es su foro oficial, donde encontrará toda la ayuda que necesite, podrá seguir los pasos de los proyectos de la comunidad y estará en contacto con los nuevos avances del lenguaje. Tampoco hay que olvidar la wiki, punto de referencia obligatorio para consulta de documentación.

***Tutorial de Trinit** (http://www.trinit.es/tutoriales ) : Muy buen tutorial introductorio a Bennu. De un nivel más básico que este texto, es ideal para introducir de forma rápida y atractiva el lenguaje Bennu a un público profano en la materia, y conseguir así que éste se interese posteriormente por el mundo Bennu. Incluye un tutorial sobre el módulo Bennu3D.

***Página de DIV Francia** (http://div.france.online.fr) : Interesante portal que aún hoy contiene buenos tutoriales y ejemplos de código. En concreto, aloja un tutorial para hacer un juego de carreras en scroll con la pantalla dividida muy didáctico

***Página de Gpwiki** (http://www.gpwiki.org): Más en general, esta página contiene toneladas de información sobre la programación de videojuegos (en ningún lenguaje en concreto). Desde ideas de cómo programar un RPG hasta cómo desarrollar un motor propio de física, en esta web hay ubicados un montón de artículos que son oro puro para el desarrollador de videojuegos.

# CAPITULO 11

## Códigos y trucos variados

Este capítulo pretende ser una miscelanea, un cajón de sastre donde pueden tener cabida los más variados temas sobre la programación de Bennu: trucos, ideas, artículos, sugerencias,etc. No pretende tener ninguna coherencia interna como los capítulos anteriores: simplemente que aporte una referencia rápida de consulta para aquel aspecto o duda concreta que nos pudiera surgir a la hora de programar. Tampoco pretende ser un resumen exhaustivo: la mayoría de información recogida y recopilada aquí se ha extraido del foro de Bennu (o de los códigos de ejemplo del BennuPack), por lo que en cualquier caso es ese lugar el más idoneo para preguntar las dudas y problemas, y donde se aportarán las soluciones más eficaces y actualizadas.

### *Creación de combos de teclas

Si has jugado alguna vez a un juego de luchas tipo “StreetFighter” o “MortalKombat”, seguro que sabrás que existen en esta clase de juegos determinadas combinaciones de teclas que el jugador ha de pulsar rápidamente para lograr realizar una acción especial, como un patada más potente, o recargar energía,etc. Estas combinaciones de teclas son los llamados combos, y la gracia de estos juegos recae en parte en que el jugador sepa realizar los combos más rápido y mejor que su contrincante, para tener más posibilidades de ganarle. Normalmente los combos son combinaciones de teclas más o menos difíciles (más difíciles será realizarlas si el jugador obtiene más beneficio) que se deberán realizar en un tiempo máximo, ya que si el jugador se demora demasiado intentando ejecutar el combo, éste no se considerará realizado correctamente.

En este apartado veremos un código de ejemplo de implementación de un combo. En concreto, el combo será pulsar una detrás de otra seis teclas : _up,_down,_up,_down,_right,_right,_down. Y entre tecla y tecla la pulsación no deberá demorarse más de 3 décimas o si no se considerará que el combo se ha deshecho.

Mientras no logremos ejecutar el combo, en pantalla aparecerá una bola que se moverá con los cursores. También aparece en pantalla un marcador que enseña si se ha pulsado alguna de las teclas que forman el combo (aparece el código numérico de la tecla concreta: por ejemplo, la tecla _up se corresponde con el número 72). Si se logra conseguir un combo, verás que en el marcador, evidentemente, aparecerán que todas las teclas del combo han sido pulsadas, y se mostrará el efecto del combo, que es rebentar la bola en bolitas pequeñas que salen disparadas por todos los ángulos.

```
/*Ejemplo de combo. Se utiliza un patron de teclas en un array que representa la
combinaci¢n del combo. Se utiliza otro array (tabla) donde se guardan las teclas que se
pulsan, funciona como una cola, de manera que las nuevas teclas eliminan a las más
antiguas. Se controla el tiempo entre las teclas, si es más de 30 centésimas se borra el
array.*/

import "mod_map";
import "mod_key";
import "mod_screen";
import "mod_text";
import "mod_timers";
import "mod_grproc";
import "mod_proc";
import "mod_rand";
import "mod_video";

Global
    int tiempo;    //guarda el tiempo entre teclas
    int cola[6];    //aqui se guardan las teclas
// el patrón de teclas para el combo
    int combopatron[6]=_up,_down,_up,_down,_right,_right,_down;
```

```
    int comboflag=0;   // cuando valga 1, habrá combo
End

Local
    Int n;
End

process main()
Begin
    set_mode(320,200,32);
    define_region(1,0,0,320,200);
    write(0,20,10,0,"arr, aba, arr, aba, der, der, aba");
    For(n=0;n<7;++n)
        write_var(0,20+n*30,30,0,cola[n]);
    End

    juego();
    While(get_id(Type juego));
        Frame;
    End
    let_me_alone();
End


Process juego()
Private
    Int arriba, abajo, derecha, izquierda;
end
Begin
    graph=new_map(40,40,32);map_clear(0,graph,rgb(0,255,0));
    x=160;
    y=100;
    Loop
        If(key(_esc)) break; End

        If(key(_up) && y>20)
            y=y-3;
            If(!arriba)
                controltiempo();
                If(coladato(_up)) haz_combo(x,y); End
                arriba=1;
            End
        End
        If(!key(_up)) arriba=0; End

        If(key(_down) && y<180)
            y=y+3;
            If(!abajo)
                controltiempo();
                If(coladato(_down)) haz_combo(x,y); End
                abajo=1;
            End
        End
        If(!key(_down)) abajo=0; End

        If(key(_right) && x<300)
            x=x+5;
            If(!derecha)
                controltiempo();
                If(coladato(_right)) haz_combo(x,y); End
                derecha=1;
            End
        End
        If(!key(_right)) derecha=0; End

        If(key(_left) && x>20)
            x=x-5;
            If(!izquierda)
```

```
                controltiempo();
                If(coladato(_left)) haz_combo(x,y); End
                izquierda=1;
            End
        End
        If(!key(_left)) izquierda=0; End

        Frame;

    End
End


/* Mete un dato en la cola. Lo hace al final,pero  se podría hacer al revés y que entrara
en el primero. Compara el patrón con la cola para ver si hay combo. Devuelve 1 en ese
caso y pone a 1 la variable "comboflag". Devuelve cero en caso contrario. Bastaría con la
variable o con el valor devuelto pero asi tienes mas posibilades para procesar el
combo.*/
Process coladato(dato)
Begin
    For(n=0;n<6;++n)//muevo los datos adelante para hacer un hueco
        cola[n]=cola[n+1];
    End
    cola[6]=dato;   // meto el dato nuevo
    For(n=0;n<7;++n)  // comparo
        If(cola[n]!=combopatron[n]) Return(0); End /*había alguno distinto*/
    End
    comboflag=1; //Combo!!
End


// controla el tiempo entre teclas y borra la cola si es mucho
Process controltiempo()
Begin
    If((timer[9]-tiempo)>30) // 30 centesimas
        For(n=0;n<7;++n) cola[n]=0; End
    End
    tiempo=timer[9]; // nuevo tiempo
End


//Realiza el efecto del combo, que es crear 10 bolitas pequeñas
Process haz_combo(x,y)
Begin
    For(n=0;n<10;n=n+1)
        bolita(x,y,rand(0,360));
    End
    comboflag=0;
End


/*Muestra el efecto visual del combo, que es que las bolitas salgan disparadas en
direcciones diferentes*/
Process bolita(x,y,angle)
Begin
    angle=angle*1000;
    graph=new_map(40,40,32);map_clear(0,graph,rgb(0,255,0));
    size=50;
    While(!out_region(id,1))
        advance(5);
        Frame;
    End
End
```

## *Cómo redefinir el teclado

Una de las características que más a menudo se suele olvidar en la programación de un videojuego pero que es bastante importante, es la opción de redefinir el teclado. Muchos juegos traen una serie de teclas por defecto que son las que todos tienen que usar para jugar. A menudo están elegidas de una manera adecuada, pero puede ser que a alguien no le guste o le resulte incómodo. La solución es dar al jugador la opción de escoger sus propias teclas para jugar. Además, resulta muy fácil de implementar.

Lo primero que hay que saber para redefinir el teclado es cómo funciona. Ya sabemos que cada tecla tiene asignado por Bennu un valor numérico determinado. Ojo, recuerda que no estamos hablando de la tabla ASCII, la cual es estandar para todos los PC y afecta a los caracteres (por lo que “a” y “A” tienen códigos ASCII diferentes), sino que estamos hablando de que Bennu asocia un número a cada tecla del teclado (por lo cual “a” y “A”, al estar en la misma tecla, tendrán el mismo código).

El programa lo único que tendrá que hacer es detectar el código que le viene y reacciona en consecuencia. Por ejemplo, el código de la tecla "a" es el 30, por lo que ya sabemos que es lo mismo poner *key(_a)* o *key(30)* –de hecho, “_a” no es más que una constante predefinida por Bennu para ayudarnos a no tener que acordarnos de cuál era el código de la tecla “a”/”A”. Y aquí está la clave de todo el misterio.

Una vez que conocemos la naturaleza numérica de las teclas la solución al problema es usar variables a la hora de detectar si el jugador ha pulsado las teclas adecuadas. Si creamos una variable llamada *arriba* y le damos a esa variable el valor numérico que corresponda a la tecla que queremos, luego comprobamos si ha sido pulsada con *key(arriba)*. Si en cualquier momento variamos el valor de *arriba* se cambiará la tecla que responde a esa acción.

Ahora el problema se presenta en qué valores guardar en la variable. Se pueden poner los valores desde programación, pero eso implicaría tener que recompilar el juego cada vez que se cambie la configuración del teclado, además de que tendríamos que pasar el código a la gente. La gente que no se dedique a programar muchas veces ni se molestará en investigar cómo hacer para cambiar las cosas, por muy bueno que sea el juego. Por lo tanto, no es una buena opción. Por lo tanto, habrá que darle la opción al usuario desde el propio programa para que haga los cambios que desee. Bueno, pues montamos el típico menú desde donde hacerlo. Pero, ¿cómo nos las arreglamos para detectar qué tecla ha elegido?

La solución más directa sería poner un *IF (key(...)) variable=...; END* para cada tecla que podamos elegir. En principio eso significaría muuuuchas líneas de código y sería bastante engorroso. Hay otra solución más elegante: ***SCAN_CODE***. Ésta es una variable global entera predefinida ya vista anteriormente, que guarda el código numérico de la última tecla que ha sido pulsada por el jugador (con buffer para cuando hay muchas y evitar su pérdida), Por lo tanto, comprobando su valor podríamos saber qué tecla ha elegido el jugador y asignarla a la variable. De esta manera, con poco código tendremos la posibilidad de elegir cualquier tecla.No confundir el uso de ***SCAN_CODE*** con el de la función **key()***:* esta función nos indica simplemente si la tecla con el código X está o no pulsada en el frame actual.

Y ¡ojo!, no confundir el significado de la variable ***SCAN_CODE*** con el de la variable ***ASCII***: la primera admite valores que representan teclas del teclado,la segunda admite valores que representan caracteres dentro de la tabla ASCII: la letra “a” y “A” se escriben con la misma tecla, por lo que ***SCAN_CODE*** valdrá lo mismo en ambos casos, pero ***ASCII*** no, porque “a” tiene un código ASCII y “A” tiene otro. Cuidado.

Naturalmente, se pueden usar truquillos como usar arrays o structs en vez de variables sueltas para guardar los códigos de las teclas, pero eso ya es una cuestión a parte. El método de todas formas tiene una pega: el uso de ***SCAN_CODE*** es más lento que usar la función *key()*. Por lo tanto, no es adecuado para usarlo en tiempo real. En el menú no importa mucho si hay que darle un par de veces para que se entere de la tecla, pero si intentas dar ese salto que te va a llevar a la salvación y el ordenador se niega a responder a nuestra órdenes con presteza... Creo que me entendéis.

Como ejemplo de lo dicho, aquí tienes un código que redefine la tecla para mover un gráfico a la derecha o a la izquieda. Para que funcione debes crear una imagen PNG llamada “raton.png”

```
import "mod_map";
import "mod_key";
import "mod_text";
import "mod_proc";
import "mod_video";
```

```
Process Main()
Private
   Int derecha;
   Int izquierda;
end
Begin
   set_mode (320,240,32);
   // Se define la tecla para ir a la derecha
   write(0,0,0,0,"Pulse tecla para derecha");
   Repeat
      derecha=scan_code;
      Frame;
   Until (derecha<>0)

   // Se define la tecla para ir a la izquierda
   delete_text(0);
   write(0,0,0,0,"Pulse una tecla para izquierda");
   Frame(500);
   Repeat
      izquierda=scan_code;
      Frame;
   Until (izquierda<>0)

   delete_text(0);
   write(0,0,0,0,"Pulse esc para terminar");

   // Se pone el gráfico en pantalla
   x=160; y=120;
   graph=new_map(20,20,32);map_clear(0,graph,rgb(255,255,255));

   Loop
      // Se mueve el gráfico
      If (key(derecha)) x++; End
      If (key(izquierda)) x--; End

      If (key(_esc)) exit(); End
      Frame;
   End
End
```

Tal como ya comentamos, recordar que existe otra variable global predefinida relacionada con el teclado interesante, aparte de ***SCAN_CODE***, llamada ***SHIFT_STATUS.*** Esta variable almacena el estado de las teclas de control; es decir: contiene el estado de las diferentes teclas de control (CTRL, SHIFT, ALT). Según su valor, sabremos si se está pulsando las siguientes teclas (además de cualquier otra tecla adicional recogida por ***SCAN_CODE*** o la función *key()*. Sus valores pueden ser los siguientes (y la suma de éstos para detectar pulsaciones simultáneas de varias teclas de control)

- 1: Se mantiene pulsada la tecla SHIFT (MAYUS) izquierda
- 2: Se mantiene pulsada la tecla SHIFT (MAYUS) derecha
- 4: Se mantiene pulsada la tecla CONTROL (cualquiera de ellas)
- 8: Se mantiene pulsada la tecla ALT (cualquiera de ellas)

***Cómo hacer que nuestro videojuego (programado en PC) pueda ejecutarse en la consola Wiz usando sus mandos**

Bennu tiene una versión oficial para la consola GP2X Wiz. Evidentemente, el lenguaje es el mismo con el que se programan los videojuegos para PC Pero hay un aspecto que es forzosamente diferente: las consolas no tiene teclado (o al menos, tal como lo conocemos en un PC). Es decir, la función **key()** no funciona en la Wiz. ¿Cómo podríamos entonces programar un videojuegos sin tener que hacer dos versiones, una para PC y otra para consola, debido a esta diferencia de comportamiento? Pues utilizando un módulo que abstraiga estas diferencias de hardware al programador y haga que éste pueda acceder de la misma manera ya sea al teclado del PC como a las teclas de la consola, de forma totalmente transparente para el programador.

A continuación se presenta el módulo (escrito en Bennu) que realiza esta labor. El código siguiente simplemente lo tendrás que incorporar a los ficheros de tu proyecto, y ya podrás tener una única manera de acceder a los teclados de ambos tipos de máquinas, sin preocuparte de si el juego está funcionando en PC o en la Wiz. Este módulo lo llamaremos "jkeys.lib", pero se puede llamar como quieras. Fíjate que depende de un módulo oficial de Bennu del cual no hemos hablado: "mod_joy", responsable del manejo de joysticks y dispositivos de entrada similares.

```
import "mod_key";
import "mod_joy";
import "mod_proc";

/* ------------------------------------------------------------------------ */

const
    _JKEY_UP        =  0 ;
    _JKEY_UPLEFT    =  1 ;
    _JKEY_LEFT      =  2 ;
    _JKEY_DOWNLEFT  =  3 ;
    _JKEY_DOWN      =  4 ;
    _JKEY_DOWNRIGHT =  5 ;
    _JKEY_RIGHT     =  6 ;
    _JKEY_UPRIGHT   =  7 ;
    _JKEY_MENU      =  8 ;
    _JKEY_SELECT    =  9 ;
    _JKEY_L         = 10 ;
    _JKEY_R         = 11 ;
    _JKEY_A         = 12 ;
    _JKEY_B         = 13 ;
    _JKEY_X         = 14 ;
    _JKEY_Y         = 15 ;
    _JKEY_VOLUP     = 16 ;
    _JKEY_VOLDOWN   = 17 ;
    _JKEY_CLICK     = 18 ;

    _JKEY_LAST      = 19 ;

end

/* ------------------------------------------------------------------------ */

global
    int jkeys[_JKEY_LAST];
    int jkeys_state[_JKEY_LAST];
end

/* ------------------------------------------------------------------------ */

function jkeys_set_default_keys()
begin
    jkeys[ _JKEY_UP        ] = _UP ;
    jkeys[ _JKEY_UPLEFT    ] = 0 ;
    jkeys[ _JKEY_LEFT      ] = _LEFT ;
    jkeys[ _JKEY_DOWNLEFT  ] = 0 ;
    jkeys[ _JKEY_DOWN      ] = _DOWN ;
    jkeys[ _JKEY_DOWNRIGHT ] = 0 ;
    jkeys[ _JKEY_RIGHT     ] = _RIGHT ;
    jkeys[ _JKEY_UPRIGHT   ] = 0 ;
    jkeys[ _JKEY_MENU      ] = _ESC ;
    jkeys[ _JKEY_SELECT    ] = _ENTER ;
    jkeys[ _JKEY_L         ] = 0 ;
    jkeys[ _JKEY_R         ] = 0 ;
    jkeys[ _JKEY_A         ] = _A ;
    jkeys[ _JKEY_B         ] = _D ;
    jkeys[ _JKEY_X         ] = _S ;
    jkeys[ _JKEY_Y         ] = _W ;
    jkeys[ _JKEY_VOLUP     ] = 0 ;
    jkeys[ _JKEY_VOLDOWN   ] = 0 ;
    jkeys[ _JKEY_CLICK     ] = 0 ;
```

```
end

/* ------------------------------------------------------------------------- */

process jkeys_controller()
private
    i;
begin

    signal_action( s_kill, s_ign );
    signal( type jkeys_controller, s_kill );
    signal_action( s_kill, s_dfl );

    loop
        for ( i = 0; i < _JKEY_LAST; i++ )
            if ( jkeys[i] )
                jkeys_state[i] = key( jkeys[i] );
            else
                jkeys_state[i] = 0;
            end
        end

        if ( os_id == OS_GP2X_WIZ && joy_number() ) /* Wiz */
            jkeys_state[ _JKEY_UP        ] |= joy_getbutton( 0, _JKEY_UP        );
            jkeys_state[ _JKEY_UPLEFT    ] |= joy_getbutton( 0, _JKEY_UPLEFT    );
            jkeys_state[ _JKEY_LEFT      ] |= joy_getbutton( 0, _JKEY_LEFT      );
            jkeys_state[ _JKEY_DOWNLEFT  ] |= joy_getbutton( 0, _JKEY_DOWNLEFT  );
            jkeys_state[ _JKEY_DOWN      ] |= joy_getbutton( 0, _JKEY_DOWN      );
            jkeys_state[ _JKEY_DOWNRIGHT ] |= joy_getbutton( 0, _JKEY_DOWNRIGHT );
            jkeys_state[ _JKEY_RIGHT     ] |= joy_getbutton( 0, _JKEY_RIGHT     );
            jkeys_state[ _JKEY_UPRIGHT   ] |= joy_getbutton( 0, _JKEY_UPRIGHT   );
            jkeys_state[ _JKEY_MENU      ] |= joy_getbutton( 0, _JKEY_MENU      );
            jkeys_state[ _JKEY_SELECT    ] |= joy_getbutton( 0, _JKEY_SELECT    );
            jkeys_state[ _JKEY_L         ] |= joy_getbutton( 0, _JKEY_L         );
            jkeys_state[ _JKEY_R         ] |= joy_getbutton( 0, _JKEY_R         );
            jkeys_state[ _JKEY_A         ] |= joy_getbutton( 0, _JKEY_A         );
            jkeys_state[ _JKEY_B         ] |= joy_getbutton( 0, _JKEY_B         );
            jkeys_state[ _JKEY_X         ] |= joy_getbutton( 0, _JKEY_X         );
            jkeys_state[ _JKEY_Y         ] |= joy_getbutton( 0, _JKEY_Y         );
            jkeys_state[ _JKEY_VOLUP     ] |= joy_getbutton( 0, _JKEY_VOLUP     );
            jkeys_state[ _JKEY_VOLDOWN   ] |= joy_getbutton( 0, _JKEY_VOLDOWN   );
            jkeys_state[ _JKEY_CLICK     ] |= joy_getbutton( 0, _JKEY_CLICK     );

jkeys_state[_JKEY_UP ] |= jkeys_state[ _JKEY_UPLEFT  ] | jkeys_state[ _JKEY_UPRIGHT  ];
jkeys_state[_JKEY_DOWN]|= jkeys_state[_JKEY_DOWNRIGHT ] | jkeys_state[ _JKEY_DOWNLEFT ];
jkeys_state[_JKEY_LEFT ] |= jkeys_state[ _JKEY_UPLEFT ] | jkeys_state[ _JKEY_DOWNLEFT ];
jkeys_state[_JKEY_RIGHT] |= jkeys_state[ _JKEY_UPRIGHT ]| jkeys_state[ _JKEY_DOWNRIGHT ];
        end
        frame;
    end
end
```

Bien, pero: ¿cómo se usa en nuestro videojuego este módulo de abstracción de dispositivos de entrada? He aquí un ejemplo autoexplicativo de cómo utilizarlo. Puedes fijarte por ejemplo que la tecla “A” de la consola se mapea (“se hace equivalente”) a la tecla “A” del teclado de PC, y así con todas.

```
import "mod_map";
import "mod_screen";
import "mod_say";
import "mod_video";

#include "jkeys.lib"

Process Main()
Begin
    jkeys_set_default_keys(); //Necesario escribirlo al principio del programa
    jkeys_controller();       //Necesario escribirlo al principio del programa
```

```
    set_mode(320,240,32);
    graph = new_map(32,32,32); map_clear(0,graph,rgb(128,0,0));
    x = 160;y = 120;

    repeat
        IF (jkeys_state[_JKEY_UP   ] && y > 0        ) y-=4; end
        IF (jkeys_state[_JKEY_DOWN ] && y < 240 - 16 ) y+=4; end
        IF (jkeys_state[_JKEY_LEFT ] && x > 0        ) x-=4; end
        IF (jkeys_state[_JKEY_RIGHT] && x < 320 - 16 ) x+=4; end

        if(jkeys_state[_JKEY_A]) map_clear(0,graph,rgb(128,0,0)); end
        if(jkeys_state[_JKEY_B]) map_clear(0,graph,rgb(0,128,0)); end
        if(jkeys_state[_JKEY_X]) map_clear(0,graph,rgb(0,0,128)); end
        if(jkeys_state[_JKEY_Y]) map_clear(0,graph,rgb(128,128,128)); end
        if(jkeys_state[_JKEY_SELECT]) map_clear(0,graph,rgb(255,255,255)); end

        FRAME;
    until(jkeys_state[_JKEY_MENU])
End
```

## *Cómo desarrollar un formato propio de imagen, de uso exclusivo en nuestros programas

A continuación se presenta un código fuente que, aún no siendo excesivamente útil en el día a día, no deja de ser muy interesante. En él se hace uso de dos funciones, "imgtxt()" y "txt2img()" el uso de las cuales se puede trasladar a cualquier otro programa sin problemas.

La primera función se encarga de, a partir de una imagen especificada como parámetro (incluida o no dentro de un FPG), codificar su información gráfica -básicamente, el color de cada pixel- en formato texto, y grabar éste en un archivo de texto especificado también como parámetro. Es decir, "transforma" una imagen en texto. La segunda función hace el proceso inverso: a partir de un archivo de texto que contiene la información generada anteriormente por "img2txt()", reconstruye la imagen tal como era la original. Es decir, "transforma" un texto en imagen.

Una posible utilidad de este código sería ocultar los recursos gráficos usados en nuestro juego de personas indeseadas. Utilizando estas funciones propias, las imágenes podrán distribuirse con (casi) total confianza por todo el mundo en un simple formato de texto sin temor a posibles "hurtos", ya que sólo aquellos que utilicen la función correcta de decodificación (es decir, "txt2img()") podrán reconstruir la imagen y la podrán visualizar.De hecho, esta idea (aunque mejorada) es utilizada hoy en día por los juegos profesionales.

Una mejora evidente del código presentado es utilizar una codificación binaria en vez de textual, ya que el texto ocupa muchísimo espacio, con lo que unas pocas fotografías pueden aumentar de manera considerable el peso de nuestro juego, pero si entramos en el terreno de las codificaciones binarias, de hecho estaremos entrando en la creación de un nuevo formato de imagen, tema bastante complejo en sí. Puedes no obstante echarle una ojeada a un interesante artículo que explica este tema muy didácticamente: http://gpwiki.org/index.php/Alternative_Binary_Maps_and_Logic

El programa lo que hace es, de entrada, mostrar como fondo una imagen (en este caso "a.png"), y dos opciones. Si se pulsa "i", se generará en el directorio actual un archivo de texto "mapa.txt" que contendrá la codificación de la imagen. Si se pulsa "c" (¡hacerlo siempre después de "i"!), se recogerá la información guardada de "mapa.txt", se borrará el fondo de pantalla que había anteriormente y se mostrará como nuevo fondo la imagen generada a partir de la información obtenida de "mapa.txt", para comprobar que la nueva imagen es idéntica a la original.

```
import "mod_video";
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_file";
import "mod_draw";
```

```
process main()
private
          int fpg,idimagen;
          string archtexto;
/*Recogerá el identificador devuelto por la función txt2img(), el cual representa una
imagen que se visualizará como nuevo fondo de pantalla*/
          int fondoimportado;
          int idtexto;
end
begin
/*Estaría bien que los siguientes tres valores se pasaran como parámetros por la línea de
comandos, usando las variables globales predefinidas ARGV y ARGC*/
  archtexto=".\mapa.txt";
  fpg=0;
  idimagen=load_png("a.png");

  write(0,150,20,4,"1º.-Pulse 'i' para codificar la imagen a texto");
  write(0,150,40,4,"2º.-Pulse 'c' para decodificar el texto a imagen");
  put_screen(fpg,idimagen);
          loop
            if (key(_i))
                  //Aviso que la función está en marcha.
                  idtexto=write(0,150,100,4,"Procesando...");frame;
                  img2txt(archtexto,fpg,idimagen);
                 //Aviso de que ya acabó
                 delete_text(idtexto);
            end
            if (key(_c))
//Borro la imagen mostrada anteriormente para poner la cargada a partir del archivo
                 clear_screen();
                 idtexto=write(0,150,100,4,"Procesando...");frame;
/*El identificador de la imagen generada por la función cuyo contenido seha rellenado a
partir de los datos almacenados en el archivo de texto se retorna y se asigna a
"fondoimportado". A partir de aquí, se pueden hacer muchas cosas: en este ejemplo se
utiliza "fondoimportado" para ponerlo de nuevo fondo de pantalla, y así comprobar que es
la misma imagen que había al principio.*/
                 fondoimportado=txt2img(archtexto);
                 delete_text(idtexto);
                 put_screen(0,fondoimportado);
            end
            if(key(_esc)) exit(); end
            frame;
          end
end


function img2txt(string archtexto,int fpg, int idimagen)
private
  int salida;

//Los campos de esta estructura son los que se van a guardar en el archivo
  struct infimg
     int x,y; //Posición de un pixel
     int color; //Su color RGB
  end

//Variables auxiliares
   int i,j;

   int ancho,alto; //De la imagen
end

BEGIN
//Creo el archivo de texto (o lo sobrescribo)
salida=fopen(archtexto,O_write);
/*Obtengo el ancho y alto de la imagen, para recorrer los bucles anidados
convenientemente...*/
ancho=map_info(fpg,idimagen,g_width);
```

```
alto=map_info(fpg,idimagen,g_height);

//...y escribirlo al comienzo del archivo de texto, como información útil a la hora de
leerlo posteriormente
fwrite(salida,ancho);          fwrite(salida,alto);

//Recorro pixel a pixel la imagen, en sus dos dimensiones
from i = 1 to ancho;
          from j = 1 to alto;
/*Asigno los valores que me interesan a los campos correspondientes de la estructura que
se escribirá en el archivo*/
                    infimg.x=i;infimg.y=j;
/*Aquí obtengo el color de cada pixel, codificado en un único número mezcla de las tres
componentes RGB.Esto tiene un problema, y es que como la codificación única RGB depende
de la tarjeta gráfica de la máquina, la información de color guardada en el archivo es
posible que en otro ordenador diferente se visualice cromáticamente de forma diferente.
Para evitar esto, se podría guardar en la estructura en vez de un único número, el valor
de las tres componentes RGB por separado. Para ello, en vez de tener un campo INT color,
tendríamos que tener tres campos BYTE llamados por ejemplo r,g y b. Y a la hora de
obtener el color de la imagen actual, obtener sus componentes con ayuda de la función
get_rgb, de la siguiente manera get_rgb(color,&(infimg.r),&(infimg.g),&(infimg.b));*/
                    infimg.color=map_get_pixel(fpg,idimagen,i,j);
//Y lo hago: escribo esos valores en el archivo de texto
                    fwrite(salida,infimg);
          end
end
fclose(salida);
end


function txt2img(string archtexto)
private
  int entrada;
/*Esta estructura tiene los mismo campos (evidentemente) que la que se usó para escribir
el archivo*/
  struct infimg
          int x,y;
         int color;
  end
/*Imagen donde se volcará la información del fichero de texto,y que será mostrada como
fondo de pantalla en el programa principal*/
  int newimagen;
  int ancho,alto; //De la imagen
end
BEGIN

Entrada=fopen(archtexto,O_read);

/*Primero leo el ancho y alto de la imagen, que son los dos datos INT que se guardaron al
principio del archivo de texto cuando se generó.Estos datos los necesito para crear la
nueva imagen con new_map con el tamaño igual al de la imagen original.*/
fread(entrada,ancho);fread(entrada,alto);

/*Genero una imagen vacía, que se rellenará del contenido gráfico obtenido a partir del
archivo de texto*/
newimagen=new_map(ancho,alto,32);

while(!feof(entrada))
//Leo los datos de un pixel
          fread(entrada,infimg);
/*Y los traduzco gráficamente: su posición y su color. Relleno de contenido pixel a pixel
la imagen generada vacía anteriormente.Luego la mostro como fondo de pantalla (esto
último se podría cambiar...)*/
          map_put_pixel(0,newimagen,infimg.x,infimg.y,infimg.color);
end
fclose(entrada);
//Devuelvo el identificador de la imagen generada a partir del archivo de texto
return (newimagen);
```

```
end
```

## *Cómo cargar imágenes en formato BMP

Aunque ya existe una muy buena librería no oficial (la "mod_image", basada en SDL_Image), de la cual hablaremos posteriormente, que es capaz de cargar multitud de formatos de imagen, posibilitando así que Bennu pueda utilizar de entrada más tipos de gráficos aparte de los PNG, creo que es interesante como ejercicio didáctico mostrar cómo sería un código, escrito en Bennu, que fuera capaz de leer un formato por defecto no soportado. En este caso hemos elegido el formato BMP,pero podría haber sido cualquier otro del cual se conozca su estructura interna. Tal como se puede ver en http://es.wikipedia.org/wiki/Windows_bitmap (o mejor, en http://en.wikipedia.org/wiki/BMP_file_format) el formato BMP puede albergar imágenes en 8 bits ó en 24 bits (no admite transparencia, pues), e incluye una serie de campos que contienen metadatos que forman la cabecera del fichero, cuyo significado lo podemos estudiar en las urls anteriores . Conociendo la estructura de dicha cabecera, se podrá obtener la información guardada en dicho fichero, ya que esta cabecera es la que nos informará sobre cómo está guardado, estructurado y formateado el contenido que nos interesa extraer.

A continuació presento un código que es capaz de leer la cabecera de un fichero BMP, campo a campo (distinguiendo antes si es de 8 bits ó 24 bits, ya que las cabeceras son diferentes), y guardar los datos obtenidos en varias estructuras, que se utilizarán para interpretar el contenido del fichero y pintar en pantalla cada píxel de la imagen en consecuencia. Para probarlo necesitarás una imagen BMP, llamada "dibujo.bmp".

```
import "mod_key";
import "mod_screen";
import "mod_video";
import "mod_file";
import "mod_map";
import "mod_draw";

process main()
private
  int imagen;
end
begin
  set_mode(640,480) ;
  imagen=load_bmp("dibujo.bmp");
  put_screen(0,imagen);
  while (not key(_esc))
   frame;
  end
end



PROCESS load_bmp(string archivo)
PRIVATE
int mapa;

struct cabecera_bmp
          byte header[1]; // carga los caracteres B y M por separado.
          int filesize; // carga el tamaño del archivo.
          word reserved[1]; // carga los bytes reservados.
          int f_offset; // carga el offset.
end

struct info_header
          int t_cabecera;
          int ancho;
          int alto;
          word planos;
          word bits;
          int compresion;
          int t_imagen;
```

```
          int ppm_h;
          int ppm_v;
          int colores;
          int colores_imp;
end

struct pixel24; //para pixels de 24 bits
          byte b;
          byte g;
          byte r;
end

struct paleta[255]; //paleta para graficos de 8 bits
          byte b;
          byte g;
          byte r;
          byte reserved;
end
byte pixel; // para pixels de 8 bits
END

BEGIN
archivo=fopen(archivo,o_read);
fread(archivo,cabecera_bmp);
fread(archivo,info_header);

mapa=new_map(info_header.ancho,info_header.alto,32);

IF(info_header.bits==8)
          fread(archivo,paleta);
          y=info_header.alto; // esto es porque la primera línea es la de abajo
          repeat
                    x=0;
                    repeat
                              fread(archivo,pixel);
         map_put_pixel(0,mapa,x,y,rgb(paleta[pixel].r,paleta[pixel].g,paleta[pixel].b));
                              x++;
                    until(x>=info_header.ancho)
                    y--;
          until(y<0)
END

IF(info_header.bits==24)
          y=info_header.alto;
          repeat
                    x=0;
                    repeat
                              fread(archivo,pixel24);
                              if(pixel24.r!=0 or pixel24.g!=0 or pixel24.b!=0)
                         map_put_pixel(0,mapa,x,y,rgb(pixel24.r,pixel24.g,pixel24.b));
                              end
                              x++;
                    until(x>=info_header.ancho)
                    y--;
          until(y<0)
END
fclose(archivo);
return mapa;
END
```

***Cómo grabar partidas y reproducirlas posteriormente utilizando almacenamiento de datos en archivo**

En este tutorial, se pondrá en práctica el uso de archivos para almacenamiento de datos de un programa (en este caso, las diferentes posiciones que tienen dos procesos cuando se mueven durante cierto tiempo) y su posterior

recuperación. Para conseguir dicho objetivo no se utilizarán **fopen(), fwrite(), fread()** y familia, sino simplemente dos funciones equivalentes: **Save()** y **Load()**. La elección entre unas u otras es puramente personal: recuerda que **save()** viene a equivaler al trío de funciones **fopen()  -en modo escritura-/fwrite()/fclose()**, y **load()** viene a equivaler al trío **fopen() -en modo lectura-/fread()/fclose()**.

Vamos a empezar con el codigo inicial del juego, el cual consistirá en 2 imagenes que se mueven en la pantalla gracias al control del jugador 1 y al jugador 2. Empecemos escribiendo esto:

```
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_key";
import "mod_video";
import "mod_file";

process main()
Begin
          set_mode(640,480,32);
          write(0,400,290,4,"1=JUGAR");
          write(0,400,300,4,"2=VER DEMO");
          write(0,400,310,4,"3=SALIR");
          loop
                    if(key(_1))
                              delete_text(0);
                              empieza_partida();
                              break;
                    end
                    if(key(_2))
                              delete_text(0);
                              ver_una_partida();
                              break;
                    end
                    if(key(_3))
                              break;
                    end
                    frame;
          end
end

Process empieza_partida()
BEGIN
          jugador1();
          jugador2();
          loop
                    write(0,400,30,4,"PULSA ESC PARA SALIR");

               if(key(_esc)) breaK; end
                    frame;
                    delete_Text(0);
          end
          let_me_alone();
end

Process jugador1()
BEGIN
          graph=new_map(50,50,32);map_clear(0,graph,rgb(255,0,0));
          x=300;y=200;
          loop
                    if(key(_up)) y=y-2; end
                    if(key(_down)) y=y+2; end
                    if(key(_left)) x=x-2; end
                    if(key(_right)) x=x+2; end
                    frame;
          end
end

Process jugador2()
```

```
BEGIN
          graph=new_map(50,50,32);map_clear(0,graph,rgb(0,0,255));
          x=200;y=400;
          loop
                    if(key(_w)) y=y-2; end
                    if(key(_s)) y=y+2; end
                    if(key(_a)) x=x-2; end
                    if(key(_d)) x=x+2; end
                    frame;
          end
end

//Más tarde nos encargaremos de este proceso
Process ver_una_partida()
BEGIN
          loop
                    frame;
          end
end
```

Ahora vamos a grabar una partida. Necesitaremos una estructura con unos campos que contengan las variables ***X*** e ***Y*** de los jugadores (para ir guardándolas cada frame). Así que en la sección de declaraciones de variables globales escribe:

```
struct rec
          int pos_jug1x[4000];
          int pos_jug1y[4000];
          int pos_jug2x[4000];
          int pos_jug2y[4000];
end
```

Y aqui nos encontramos con la desventaja de este metodo. ¿Como sabremos si tendremos que usar 4000 o más?. Pues eso es problema tuyo. Yo aconsejaria estimar una cantidad de acuerdo a un tiempo limite de duracion que tenga el juego o, mejor, usar arrays dinámicos (es decir, listas simple o doblemente enlazadas, pilas, colas, etc). No obstante, los arrays dinámicos (y el uso que conlleva de la gestión de la memoria mediante punteros) es un concepto avanzado de programación que en este curso ni tocaremos.

Tambien vamos a necesitar una variable auxiliar que vaya recorriendo cada posicion en cada frame (enseguida lo entenderás mejor), así que crearemos la siguiente variable, en la sección global:

```
int varaux=0;
```

Esta variable irá incrementando su valor de a 1 cada frame. Para eso, justo antes de la orden frame del proceso "empieza_partida()" escribe:

```
varaux=varaux+1;
```

Y justo antes de la orden frame del proceso "jugador1()", escribe:

```
rec.pos_jug1x[varaux]=x;
rec.pos_jug1y[varaux]=y;
```

¿Qué estamos haciendo con esto? Guardar en cada frame, el valor actual de ***X*** e ***Y*** dentro de la estructura *rec*, rellenando en cada frame,de forma secuencial, cada uno de sus hasta 4000 elementos respectivos,de manera que en cada elemento se almacenen las posiciones consecutivas por las que ha pasado el jugador 1.

Es decir, suponiendo que el jugador 1 se encuentra en la posicion *x=300* y pulsa la tecla para ir a la derecha, el valor de la variable *rec.pos_jug1x[varaux]*, cuando *varaux* sea igual a 0, esta sera igual a *rec.pos_jug1x[varaux]=300;* . Cuando *varaux=1* (en el siguiente frame), tendremos *rec.pos_jug1x[varaux]=302*, cuando *varaux* valga 2 (en el siguiente frame), tendremos *rec.pos_jug1x[varaux]=304;* y asi, sucesivamente, iremos guardando las coordenadas de jugador 1.

Un detalle importante es ponerle un límite a la variable *varaux*, para que nunca sea mayor que 4000, ya

que en este caso, si el juego sigue y *varaux=4001* o superior podriamos acceder a una parte de la memoria que está fuera de la estructura que hemos creado y por tanto estarÍamos armando un gran lio. asi que debajo de la línea *varaux=varaux+1;* del proceso "empieza_partida()",agregaremos lo siguiente:

```
if(varaux>=4000) varaux=4000; end
```

Lo mismo que hemos hecho en el proceso "jugador1()", deberemos escribirlo en el proceso "jugador2()", utilizando, eso sí, rec.jug2x y rec.jug2y.Es decir, justo antes de la orden frame del proceso "jugador2()", escribe:

```
rec.pos_jug2x[varaux]=x;
rec.pos_jug2y[varaux]=y;
```

Ahora, lo único que nos queda es hacer que todo el contenido de la estructura rec pase a grabarse al disco duro en forma de fichero, y tenerlo así disponible posteriormente para su visualización. Haremos que se grabe pulsando la tecla ESC, y utilizaremos la función **save(),** tal como hemos comentado antes. Así pues, cuando pulsemos la tecla ESC guardaremos en el disco la estructura *rec* que contiene la posiciones de los jugadores. Para ello, en el proceso "empieza_partida()", dentro del bloque *if(key(_esc))/end* escribe lo siguiente (ojo, ha de ser la primera línea de todas, antes del BREAK):

```
save("partida.dem",rec);
```

Y ya sólo nos queda poder visualizar la partida previamente grabada. Para ello modificaremos el proceso "ver_una_partida()", que ahora no hace nada, y crearemos 2 procesos más que simularán al jugador 1 y 2, y harán lo mismo que hicieron éstos en la partida que se grabó. Para ello, sus valores ***X*** e ***Y*** tendran el valor de las variables *rec.pos_jug1x[varaux]*, *rec_pos_jug1y[varaux]*,etc. Llamaremos a estos procesos "rec_jugador1()" y "rec_jugador2()", y su código (junto con el código modificado de "ver_una_partida()") es éste:

```
Process ver_una_partida()
BEGIN
          load("PARTIDA.dem",rec); //cargamos el archivo
          rec_jugador1();
          rec_jugador2();
          loop
                     write(0,400,30,4,"PULSA ESC PARA SALIR");
                    if(key(_esc))
                              breaK;
                    end
                    varaux=varaux+1;
                    if(varaux>=4000) varaux=4000; end
                    frame;
                    delete_text(0);
          end
          let_me_alone();
end

Process rec_jugador1()
BEGIN
          graph=new_map(50,50,32);map_clear(0,graph,rgb(255,0,0));
          loop
//X TENDRA EL VALOR DE LA VARIABLE POS_JUG1X EN LA POSICION varaux
                    x=rec.pos_jug1x[varaux];
//Y TENDRA EL VALOR DE LA VARIABLE POS_JUG1Y EN LA POSICION varaux
                    y=rec.pos_jug1y[varaux];
                    frame;
          end
end

Process rec_jugador2()
BEGIN
          graph=new_map(50,50,32);map_clear(0,graph,rgb(0,0,255));
          loop
                    x=rec.pos_jug2x[varaux];
                    y=rec.pos_jug2y[varaux];
                    frame;
          end
```

```
end
```

## *Cómo controlar muchos individuos

En este apartado comentaremos una manera (hay miles) de solucionar el asunto de controlar distintos individuos de manera general, sin recurrir a complicados mecanismos de inteligencia artificial. No obstante, si deseas profundizar en el tema de los juegos de estrategia en tiempo real (tema que no abordaremos aquí), te aconsejo que le eches una ojeada al tutorial (en inglés) disponible en la web http://www.rakrent.com/rtsc/rtsc_basics.htm

Para controlar un ejército entero de soldados, se puede hacer que cada instancia de proceso (cada soldado individual) se encargue de detectar si tiene que hacer algo o no, pero puede resultar un poco complicado, especialmente si hay varias maneras de darle órdenes a cada individuo. Si tenemos, además, distintos tipos de individuos con su propio proceso tendremos que repetir el código en cada proceso, con todo lo que esto conlleva: posibles errores al duplicar el código, tener que cambiar todos los procesos afectados si se hace algún cambio, código más largo y menos legible,...

¿Y qué solución propongo yo? Algo tan sencillo como tener una estructura general donde guardar las distintas órdenes que recibirán los individuos. En esa estructura podemos guardar datos como la ID del proceso (muy útil en muchos casos, aunque sólo sea para destruirlo), sus coordenadas y/o número de casilla del escenario (para saber dónde está), las órdenes que tiene que cumplir (codificadas en un número: 1-> moverse ; 2-> disparar ; 3-> cavar trincheras ; ...), coordenadas de las órdenes que tiene que cumplir (a dónde se tiene que mover, qué posición tiene que bombardear,...), a qué individuo tiene que atacar (su id, su posición en la estructura,...), cuánta energía le queda,... Cualquier dato que necesite nuestro individuo. De esta manera, cada proceso no tiene más que mirar en la tabla para saber qué tiene que hacer, si le queda mucha o poca energía,etc.

En otro proceso, podremos poner todas las rutinas de control de individuos. Simplemente sabiendo el índice en la tabla del individuo al que le queremos dar órdenes podemos controlar todos los aspectos de ese individuo: le podemos decir que vaya a una posición concreta del mapa (la del ratón,...), cambiar el estado del individuo (que no se mueva ni ataque hasta que haya sido liberado, bajo los efectos de un ataque que le hace moverse más despacio,...), reducir su energía (exposición a rayos cósmicos mortales,...),... De esta manera tendremos todo el control centralizado en un único proceso, por lo que facilitaremos el desarrollo de nuevas aspectos del control, evitaremos muchos errores y abriremos más posibilidades. Por ejemplo, sería mucho más fácil hacer alguna especie de animación previa al principio de la fase (o en medio o donde queramos), ya que bastaría con ir modificando los datos de la estructura y los individuos responderán obedientemente. Y si no hemos activado el proceso encargado de darle el control al jugador no tendremos que preocuparnos por que el jugador intente mover las tropas o hacer cosas raras. Además, si creamos un nuevo tipo de tropa que se mueve dando saltos, ataca insultando al enemigo o hace cualquier otra cosa extraña no tenemos que tocar el sistema de control, simplemente tenemos que programar el proceso de manera que el individuo responda a las órdenes de la manera adecuada.

Otra posibilidad es dar órdenes a varios individuos a la vez. Se pueden guardar sus posiciones en la estructura en un pequeño array e ir variando a la vez todos las posiciones de la tabla que hagan falta. Así nos dará igual si los individuos que hemos elegido son del mismo tipo, de distinto tipo, si se pueden mover o no, todos mirarán las órdenes y su estado y actuarán en consecuencia. Todo esto unido a que cualquier proceso puede acceder en cualquier momento a estos valores para realizar cualquier tipo de comprobación o ajuste lo convierten, en mi opinión, en un buen sistema.

De todas formas el sistema tiene sus pegas. La primera es que si queremos eliminar un individuo y luego poder meter otros nuevos tendremos que andar recolocando la lista, es decir, mover todos los que estén por debajo una posición hacia arriba. Eso es fácil de hacer, pero entonces los procesos no estarán mirando la tabla en la posición adecuada. Habría que corregir eso. La manera más sencilla es hacer que el proceso compruebe si su ID y la de la posición de la tabla que está mirando son la misma y pasar a mirar en la posición superior de ahí en adelante si no lo es. Es muy importante realizar esta comprobación y ajuste antes de mirar qué órdenes tiene que cumplir el individuo en cuestión, es decir, que sea lo primero que haga el proceso dentro del LOOP. Si no lo hacemos así puede ser que los datos de la tabla ya se hayan recolocado para tapar el hueco pero el proceso mire en la posición equivocada (en realidad estará mirando las órdenes y los datos del siguiente individuo en la lista).

Otro problema importante es que el número de individuos estará limitado por el número de posiciones que le hayamos puesto a la estructura, a no ser que vayamos aumentándola de manera dinámica. Esto se puede

solucionar poniendo un número de posiciones lo suficientemente alto (muchos juegos comerciales tienen límite de número de unidades: en el Starcraft son 200). Lo malo es que si hay pocos individuos que controlar entonces estaremos desaprovechando bastante memoria, pero no es una grave pega hoy en día. Si lo hacemos de manera dinámica usaremos la memoria de una manera mucho más eficiente, pero complicaremos el código bastante.

He hecho un pequeño programa en el que se puede controlar a 4 individuos. Se selecciona cada uno presionando su número en el teclado. Entonces dejará de estar semitrasparente e irá a cualquier posición que pinchemos con el ratón (botón izquierdo). Si mientras se está moviendo pinchamos en otro sitio irá a ese nuevo sitio, aunque no haya llegado todavía al que tenía que ir (se podría haber implementado un sistema de puntos de ruta). Si mientras se está moviendo seleccionamos otro individuo le podremos dar órdenes y se moverá, pero sin afectar al movimiento del individuo original, que irá hasta donde tiene que ir. No se ha implementado la posibilidad de elección múltiple para no complicar el programa.

Y esto es todo en principio. La base de la idea es ésa y a partir de ahí lo único que tienes que hacer es particularizarla para tu caso.

```
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_key";
import "mod_mouse";

GLOBAL
   Int grafico;        // Gráfico de los bichos
   Int seleccionado;   // Qué individuo hemos seleccionado
   struct bichos[3];   //4 posiciones para 4 bichos(empezando de 0)
            int x_dest;   // Posición a la que se tienen que mover
            int y_dest;
   END
END

process main()
BEGIN
   // Se carga el gráfico de los individuos y del ratón
   grafico=new_map(30,30,32);map_clear(0,grafico,rgb(255,255,0));
   mouse.graph=new_map(10,10,32);map_clear(0,mouse.graph,rgb(255,255,255));

   // Se llama a los procesos de los individuos
   bicho(0,0,grafico,0);
   bicho(0,0,grafico,1);
   bicho(0,0,grafico,2);
   bicho(0,0,grafico,3);

   write(0,10,0,0,"Pulsa 1-4 para seleccionar el individuo");
   write(0,10,10,0,"Pulsa el ratón para hacer que un individuo se mueva");

   LOOP
      // Se selecciona el individuo correspondiente
      IF (key(_1)) seleccionado=0; END
      IF (key(_2)) seleccionado=1; END
      IF (key(_3)) seleccionado=2; END
      IF (key(_4)) seleccionado=3; END

      // Se pone el destino al pinchar con el ratón
      IF (mouse.left)
         bichos[seleccionado].x_dest=mouse.x;
         bichos[seleccionado].y_dest=mouse.y;
      END

      IF (key(_esc)) exit(); END
      FRAME;
   END
END

// Proceso que controla a los individuos
// cual: variable que indica en qué posición de la struct tiene que mirar
```

```
PROCESS bicho(x,y,graph,cual)
BEGIN
   LOOP
      // El individuo se mueve hacia su destino (si hace falta)
      IF (bichos[cual].x_dest>x) x++; END
      IF (bichos[cual].x_dest<x) x--; END
      IF (bichos[cual].y_dest>y) y++; END
      IF (bichos[cual].y_dest<y) y--; END

      // Si no está seleccionado se pone en semitrasparente
      flags=4;
      IF (seleccionado==cual) flags=0; END

      FRAME;
   END
END
```

## *Cómo crear menús de opciones

En multitud de ocasiones desearemos incluir al inicio de nuestro juego un menú con diversas opciones (“iniciar nueva partida”, “recuperar una partida anterior”, “salir”,etc), donde gracias a un gráfico que hará las veces de puntero podremos saltar mediante el teclado de una opción a otra hasta elegir la deseada.

En el capítulo dedicado a la creación de un RPG ya se expusieron las técnicas básicas para crear un menú típico, pero este aspecto en ese momento tal vez quedaba un poco oculto entre todos los demás tratados en ese tutorial, así que he considerado aconsejable tratar el tema de los menús en un apartado diferente tal como éste, donde además se verán diferentes alternativas, desde las más fáciles hasta las más sofisticadas.

Una de las posibilidades más fáciles sería ésta (un simple SWITCH que recoge las diferentes opciones posibles que el usuario puede escoger de una lista):

```
import "mod_timers";
import "mod_text";
import "mod_key";
import "mod_rand";
import "mod_screen";
import "mod_map";
import "mod_proc";
import "mod_video";

Process Main()
Private
   int idtext;
End
Begin
set_mode(320,240,32);

//Durante 3 segundos, se produce una presentación original...(se puede quitar)
timer[0] = 0;
idtext=write(0,0,0,0,"Hola");
While (timer[0]< 300)
        move_text(idtext,rand(0,320),rand(0,240));
          Frame;
End

//Después de la presentación, se carga el menú propiamente
Menu();
End


Process Menu ()
Private
          int opcion;
```

```
End
Begin
delete_text(0);
Write (0, 160, 120, 4, "Opciones de menu");
write (0, 160, 140, 4, "1.-Jugar");
write (0, 160, 160, 4, "2.-Opciones del juego");
write (0, 160, 180, 4, "3.-Salir");
Loop
          If (Key (_1))
                    //Se carga el nivel y empieza el juego
                    Fade_Off (); delete_text(all_text); Fade_On ();
                    Inicia_Juego ();
                    break;
          End
          If (Key(_2))
                    //Cambiamos las opciones del programa
          End
          If (Key(_3))
                    //Finalizo el programa
                    Exit ("JUEGO", 0);
          End
          Frame;
End
End

Process Inicia_Juego ()
private
   int grafico;
end
Begin
grafico=new_map(320,240,32);map_clear(0,grafico,rgb(255,255,0));
put_screen(0,grafico);
While (!Key (_esc))
          Frame;
End
Exit ("JUEGO", 0);
End
```

Otra posibilidad es hacer el menú utilizando el método de detección de durezas. La idea es que el cursor del ratón detecte mediante el color del botón que representa una de las opciones del menú cuál es la opción existente bajo dicho cursor. Por ejemplo, en el siguiente código, se genera dinámicamente un menú formado por cuatro rectángulos de diferentes colores y se comprueba permanentemente que el puntero del ratón colisione sobre uno o sobre otro. Siempre que se haya pulsado además el botón izquierdo, se mostrará un mensaje diferente para cada entrada del menú.

```
import "mod_map";
import "mod_video";
import "mod_mouse";
import "mod_proc";

Process Main()
 Begin
    set_mode(640,480,32);
    menu();
 End

Process menu()
 Private
   int color;
 End
 Begin
 x=100;y=100;
 graph=new_map(200,200,32);drawing_map(0,graph);
 drawing_color(rgb(255,0,0));draw_box(0,0,200,50);
 drawing_color(rgb(0,255,0));draw_box(0,50,200,100);
 drawing_color(rgb(0,0,255));draw_box(0,100,200,150);
 drawing_color(rgb(255,255,0));draw_box(0,150,200,200);
```

```
 mouse.graph=new_map(20,20,32);map_clear(0,mouse.graph,rgb(255,255,255));
 loop
        color=map_get_pixel(0,graph,mouse.x,mouse.y);
        switch(color)
                    Case rgb(255,0,0):
                    //Se podría también cambiar el mouse.graph
                              If (mouse.left)
                    //Aquí se deberá hacer lo correspondiente a la opción seleccionada
                                        delete_text(all_text);
                                        write(0,350,100,0,"INICIO");
                              End
                    End
                    Case rgb(0,255,0):
                              If (mouse.left)
                                        delete_text(all_text);
                                        write(0,350,100,0,"CREDITOS");
                              End
                    End
                    Case rgb(0,0,255):
                              If (mouse.left)
                                        delete_text(all_text);
                                        write(0,350,100,0,"OPCIONES");
                              End
                    End
                    Case rgb(255,255,0):
                              If (mouse.left)
                                        delete_text(all_text);
                                        exit();
                              End
                    End
           End
           frame;
 End
End
```

Otro ejemplo es el siguiente, donde el truco está en tener las opciones colocadas en posiciones verticales específicas, y utilizar el SWITCH con opciones que cambian no en función del color (como el caso anterior) sino en función de la posición vertical que en ese momento tenga un cursor, móvil mediante teclado.

```
import "mod_key";
import "mod_map";
import "mod_text";
import "mod_video";
import "mod_proc";

process main()
begin
set_mode(320,240,32);
cursor();
repeat
     frame;
until (key(_esc))
exit();
end

process cursor()
private
  int y_de_la_primera_opcion=20;
  int y_de_la_ultima_opcion=50;
  int distancia_entre_opciones=10;
end
begin
graph=new_map(20,20,32);map_clear(0,graph,rgb(200,100,50));
x=30;y=20;

loop
```

```
          //Si se pulsa arriba y el cursor no esta arriba del todo
          if(key(_up) and y !=y_de_la_primera_opción)
               while(key(_up))frame; end
               y=y-distancia_entre_opciones;
          end

          //Si se pulsa abajo y el cursor no esta abajo del todo
          if(key(_down) and y !=y_de_la_ultima_opción)
               while(key(_down))frame; end
               y=y+distancia_entre_opciones;
          end

          if(key(_enter))
          //Ya que cada opción esta a un valor de y (altura) distinto
               switch(y)
                    case 20:
                              delete_text(0);
                              write(0,100,100,4,"Has pulsado la primera opción");
                    end
                    case 30:
                              delete_text(0);
                              write(0,100,100,4,"Has pulsado la segunda opción");
                    end
                    case 40:
                              delete_text(0);
                              write(0,100,100,4,"Has pulsado la tercera opción");
                    end
                    case 50:
                              delete_text(0);
                              write(0,100,100,4,"Has pulsado la cuarta opción");
                    end
               end
          end
          frame;
end
end
```

Otro ejemplo, un poco más sofisticado, que utiliza regiones y muestra un efecto de máquina de escribir, es el siguiente:

```
import "mod_key";
import "mod_map";
import "mod_proc";
import "mod_text";
import "mod_screen";
import "mod_mouse";
import "mod_video";
import "mod_grproc";

global
          int opcion;
end

process main()
begin
  set_mode(640,480,32);
  menu();
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

//Código de creacion del menu de la presentacion con efecto maquina de escribir
PROCESS Menu()
BEGIN
          mouse.graph = new_map(5,5,32);map_clear(0,mouse.graph,rgb(255,0,0));
          CreaOPCIONES (0, 280, 240, 1, "Empezar el juego");
```

```
            FRAME(5000);
            CreaOPCIONES (0, 280, 260, 2, "Salir del juego");
            LOOP
                        SWITCH (opcion)
                                    CASE 1: /*Juego();*/ write(0,100,100,4,"A JUGAR!"); END
                                    CASE 2: exit(); END
                        END
                        FRAME;
            END
END


PROCESS CreaOPCIONES(fuente, x, y, region, string opTexto)
PRIVATE
            INT Ancho_Texto;
END
BEGIN
            Ancho_Texto = text_width(0, opTexto) + 10;
            OpcionMENU(30, x, y, (x + (Ancho_Texto))+2, opTexto, 1, region);
            graph = write_in_map(0, opTexto, 3);
            z = -2;
            LOOP
                        IF (mouse.x > x AND mouse.x < ((x + (Ancho_Texto))+2) AND mouse.y >
(y - text_height(0, opTexto)/2) AND mouse.y < (y + text_height(0, opTexto)/2) AND
mouse.left)
                            Opcion = region;
                        END
                        IF (son.x <> (x + (Ancho_Texto))+2)
                                    FRAME(400);
                        ELSE
                                    FRAME;
                        END
            END
END


PROCESS OpcionMENU(graph, x, y, xFinal, STRING opTexto, INT Sent, reg)
PRIVATE
            INT xt1;
END
BEGIN
            size = 20;
            xt1 = x;
            z = -3;
            LOOP
                        IF (x <> xFinal)
                                    x=x+Sent;
                                    define_region(reg, xt1, (y - text_height(0, opTexto)/2),
x-xt1, (text_height(0, opTexto)));
                        END
                        IF (collision(TYPE mouse) and mouse.left) Opcion = reg; END
                        FRAME(30);
            END
END
```

Finalmente, incluimos aquí (por curioso a la par que interesante) el código de ya no un menú propiamente dicho, sino de una especie de introducción animada a nuestro juego (o a un determinado nivel). En lugar de los cuadrados verdes y lilas, se pueden colocar letras ó simbolos especiales.

```
import "mod_key";
import "mod_map";
import "mod_proc";
import "mod_video";

process main()
begin
  set_mode(640,480,32);
```

```
  from x=50 to 600 step 100;
          proceso1(x);
          frame(150);
  end
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

PROCESS proceso1(x)
PRIVATE
          INT Dire = 2;
          INT valFRAME=5;
END
BEGIN
        graph=new_map(50,50,32);map_clear(0,graph,rgb(130,230,60));
          z = -1;
          y= 140;
          size = 400;
          LOOP
               IF (size >= 102) size=size-2; END
               IF (size == 102) map_clear(0,graph,rgb(200,100,240)); valFRAME = 100; END
               IF (size == 100)
                      IF (y > 136) Dire = -2; END
                      IF (y < 128 and y > 120) Dire = 2; END
                      y = y+ Dire;
               END
               FRAME(valFRAME);
          END
END
```

## *Tutorial para dar los primeros pasos en el desarrollo de un juego de plataformas

Un juego de plataforma típico es un juego 2D donde el jugador maneja un personaje que debe ir avanzando por la pantalla horizontalmente (normalmente de izquierda a derecha) a través de un escenario, el cual se moverá en scroll horizontal de forma conveniente, y donde aparecen diferentes obstáculos que el protagonista ha de sortear, saltándolos por ejemplo, si no quiere perder vida o puntos. También se pueden encontrar premios por el camino, y en general está compuesto por varios niveles, a los que se llega después de haber completado el anterior. Esto juegos se llaman así porque los escenarios suelen constar de las llamadas plataformas, que son superfícies elevadas respecto el suelo por donde el protagonista puede caminar, previo salto, para sortear un enemigo o conseguir un premio.La gracia del juego es ir cambiando de plataformas según aparezcan en el escenario y según la necesidad.Un ejemplo podría ser el Mario Bros.

En este tutorial haremos un juego de plataformas de un solo nivel, pero es fácilmente ampliable. Primero de todo dibuja el fondo scrolleable (que incluya el dibujo de las plataformas) sobre el cual nuestro personaje caminará. El fondo puede ser como tú quieras: con nubes, montañas, edificios,etc, aunque ten en cuenta que como en nuestro caso haremos el juego con una resolución de 640x480, la altura de la imagen del escenario será 480, ya que no se va a mover verticalmente; en cambio, la anchura de la imagen conviene que sea más o menos elevada (bastante mayor que 640 píxeles) para dar la sensación de que el escenario es largo y diferente.Guárdalo en un archivo FPG con el código 20.

En este tutorial no lo haremos, pero en vez de utilizar un único gráfico de scroll que contiene el dibujo de fondo y el dibujo de las plataformas, podrías utilizar dos gráficos de scrolls diferentes: uno sería el gráfico de segundo plano del scroll (el 4º parámetro de **start_scroll()**) que representaría un paisaje más o menos bonito, y el otro sería el gráfico del primer plano del scroll (el 3º parámetro de **start_scroll()**), el cual contendría únicamente el dibujo de las plataformas, y el resto sería una gran zona transparente. Si lo hiciéramos de esta manera, tendríamos más libertad: podríamos cambiar el paisaje sin cambiar las plataformas y viceversa, o (con el uso adecuado del último parámetro de **start_scroll()**) podríamos dejar el paisaje inmóvil mientras se estuviera haciendo el scroll en los dibujos de las plataformas -creando un efecto interesante-,etc.

Dibuja también el mapa de durezas correspondiente al escenario. Para ello, crea una imagen de igual

tamaño que el escenario (para no hacerla de nuevo, puedes copiarla con otro nombre), y pinta de un color concreto (nosotros usaremos el rojo puro) las partes cuyas coordenadas coincidan con las zonas del escenario donde el personaje podrá caminar: procura reseguir exactamente la forma de las plataformas y precipicios. Después pinta de otro color (negro, por ejemplo) todo el resto de la imagen entera. Guárdalo en el FPG con el código 27.

Y dibuja también nuestro personaje, de perfil, mirando hacia la derecha, de unos 70x40 píxeles. Guárdalo en el FPG con el código 001.Además, tal como hicimos con el tutorial del RPG, dibujaremos varias imágenes extra del personaje ligeramente diferentes para simular el efecto de caminar.En concreto, crearemos dos imágenes más, con los códigos 002 y 003.

El código del programa principal podría empezar siendo algo parecido a esto:

```
import "mod_video";
import "mod_map";
import "mod_scroll";
import "mod_wm";

global
          int idescenario;
end

process main()
private
end
begin
          set_mode(640,480,32);
          set_fps(60,1);
          set_title("Mi juego de plataformas");
          idescenario=load_fpg("plat.fpg");
          start_scroll(0,0,20,0,0,1);
          loop
                    frame;
          end
end
```

Vamos ahora a crear nuestro personaje, en el extremo izquierdo de la pantalla, encima del suelo. Para ello, justo antes del bucle LOOP escribimos:

```
player();
```

Y crearemos el proceso “player()”, el cual nos mostrará el personaje, de momento sin animación. Aprovechando, también haremos que éste tenga la posibilidad de saltar.

```
Process player ()
BEGIN
         ctype=c_scroll;
/*Esto ya sabes que bloquea la cámara para que siga a este proceso por pantalla. Es todo
lo que hay que hacer para realizar el scrolling  automáticamente*/
         scroll.camera=id;
         x=600;
         y=330; //Cambia los valores de las coord iniciales según tus  necesidades
         Graph=1;
         LOOP
                  If (key (_right))
                           x=x+5;
                           flags=0;
                  End

                  If (key (_left))
                           x=x-5;
                           flags=1;
                  End

                  FRAME;
         End
END
```

Si quisiéramos añadirle algo más de sofisticación al movimiento horizontal de nuestro personaje, podríamos incluir algún tipo de aceleración. Por ejemplo, el bucle Loop/End del proceso anterior podría ser escrito así:

```
LOOP
  IF (key(_right)  AND SPEED<8)
      SPEED=SPEED+2;        //Incrementa la velocidad a la derecha
      flags=0;
  ELSE
      IF (key(_left) AND SPEED>-8)
          SPEED=SPEED-2;       //Incrementa la velocidad a la izquierda
          flags=1;
      ELSE
          if(not key(_LEFT) and not key(_RIGHT))
                IF (SPEED>0)  //Si no se pulsa ninguna tecla, hay una desaceleración
                     SPEED--;
                END
                IF (SPEED<0)  //O lo contrario
                    SPEED++;
                END
          end
      END
  END

  last_speed=speed;
  IF(last_speed<>0)
       incx=last_speed/ABS(last_speed);
       WHILE(last_speed<>0)
         x=x+incx; // Se produce el movimiento del carácter según la velocidad calculada
         last_speed=last_speed-incx;
      END
  END
  frame;
END
```

donde todas las variables que aparecen (*speed, incx,* etc) son privadas del proces. No obstante, esta modificación no la implementaremos, para mantener lo máximo posible la sencillez de nuestro ejemplo.

Lo que sí que vamos a hacer es añadir al proceso "player()" una sección de declaraciones de variables privadas (enteras).En concreto, inicalizaremos las siguientes variables así:

```
Private
         Int xx=6;
         Int yy=12;
         Int iy=1;
end
```

Estas variables serán usadas para hacer que nuestro personaje salte. Añadiremos ahora pues la tecla de salto y la capacidad de saltar hacia la derecha.Justo antes de la línea del Frame del proceso "Player()" escribimos:

```
If (key(_s) AND FLAGS==0)
         REPEAT
                  y=y-(yy-iy);
                  x=x+xx;
                  FRAME;
                  iy=iy+1;
         UNTIL (map_get_pixel(0,27,x,y)==rgb(255,0,0))
         iy=1;
End
```

Esto hace que el jugador salte hacia la derecha de forma realista cuando se pulse la tecla "s". Fíjate que el truco del realismo del salto está en que al principio la coordenada ***Y*** del proceso decrece muy rápido –y la ***X*** también-, pero poco a poco la coordenada ***Y*** va decreciendo a cada frame menos y menos hasta llegar un punto donde ya no decrece y comienza a crecer poco a poco a más velocidad: para verlo mejor observa y estudia qué valores tiene ***Y*** en cada iteración del repeat/until en función de lo que vale la variable *YY* –que no cambia- y lo que vale *IY* –que aumenta a

cada iteración).El personaje continuará cayendo hasta que alcance una zona (en el mapa de durezas) que tenga el color rojo puro. Se supone que el suelo en el mapa de durezas lo habrás pintado de rojo, tal como hemos dicho ¿no?, aunque en el juego se vea un precioso prado verde…

Para hacer que nuestro personaje salte hacia la izquierda, tendremos que añadir otro if antes de la línea del frame del proceso "player()":

```
If (key(_s) AND FLAGS==1)
        REPEAT
                y=y-(yy-iy);
                x=x-xx;
                FRAME;
                iy=iy+1;
        UNTIL (map_get_pixel(0,27,x,y)==rgb(255,0,0))
        iy=1;
End
```

La única cosa que ha cambiado es que estamos restando valores a ***X*** en vez de sumarlos. Y ya está.

Incluso podrías hacer que el jugador tuviese la posibilidad de pulsar otra tecla con un salto megaultra que pudiera llegar más alto, o más lejos. Simplemente habría que cambiar los valores de las variables *yy,xx* e *iy*. Si has dibujado plataformas flotantes, asegúrate que nuestro personaje puede alcanzarlas con la potencia de salto que le hemos asignado.

Ya es hora de incorporar la animación del caminar. Utilizaremos el mismo método que usamos por ejemplo en el tutorial del matamarcianos para generar las explosiones (puede haber otros, pero creo que este es el más fácil).Añade a la sección de declaraciones privadas del proceso "player()" la variable entera *cont* iniciándola a 0, y cambia los bloques *if(_right)/end* y *if(_left)/end* del proceso "player()" por estos otros:

```
if(key(_right))
        x=x+5;
        flags=0;
        for(cont=3;cont>=1;cont=cont-1)
                graph=cont;
                frame;
        end
end
```

Y

```
 if(key(_left))
        x=x-5;
        flags=1;
        for(cont=3;cont>=1;cont=cont-1)
                graph=cont;
                frame;
        end
end
```

Quizás te haya sorprendido que el FOR es decreciente. Esto lo he hecho así para que la última imagen que se vea al salir de la última iteración (y por tanto, la imagen que volverá a ser la de nuestro personaje quieto) sea siempre la misma, la número 1.

Un detalle importante, recuerda, es poner la orden FRAME dentro del for, porque si no el bucle se realizaría entero pero no se vería nada porque sólo se mostraría en pantalla el resultado final (la imagen 1) en el FRAME del final del proceso. Si queremos "hacer un seguimiento" iteración a iteración de los cambios en la variable ***GRAPH***, tenemos que forzar a que el fotograma se lance contra la pantalla a cada momento.

Ahora lo que haremos será añadir un precipio por donde el jugador podrá caer si no va con cuidado. En este caso haremos el precipicio directamente en el suelo. El funcionamiento para precipicios situados en plataformas elevadas es muy parecido: sólo hay que cambiar la condición que define cuándo nuestro personaje para de caer: si desaparece por debajo de la ventana de juego (el supuesto que haremos) o si cae en una plataforma inferior (supuesto que no hemos desarrollado pero fácil de ver: simplemente hay que utilizar la función **map_get_pixel()** para ver si en la

caida el personaje choca con una plataforma dada).

Para hacer que el personaje caiga si camina por el suelo, tendremos que eliminar alguna porción del suelo de color rojo del mapa de durezas y pintarlo como el resto, negro. Una vez hecho esto, verás que nuestro personaje, si salta sobre el precipicio, cae infinitamente y nos quedamos con el bonito escenario en pantalla sin poder hacer nada de nada.Lo que nosotros queremos es que una vez que caiga, nuestro personaje muera. Para eso, tendremos que añadir el código necesario en el proceso principal, porque aunque podríamos decir a un proceso que se matara a sí mismo (con la orden *signal(id,s_kill);*), es mejor hacerlo desde el proceso principal para tener este asunto centralizado y así evitar posibles errores posteriores derivados de despistes (querer utilizar un proceso cuando se ha suicidado antes, y cosas así). Por tanto,añadiremos la siguiente porción de código dentro del bloque Loop/End del proceso principal, justo antes del Frame;.

```
/*La máxima coordenada Y es 480, según nuestra resolución. Añadimos unos píxeles más para
que el personaje logre desaparecer de la pantalla por completo. Sería buena idea hacer
una constante que fuera la resolución vertical y así poner en este if una condición
dependiente de esa constante: sería mucho más fácil cambiar los valores si hubiera que
hacerlo alguna vez.*/
if(son.y>=500)
        signal(son,s_kill);
        player();
end
```

Fíjate que lo que hacemos es, en cada iteración del programa principal, comprobar si el proceso hijo (“player()” es el único hijo que hay del programa principal y por eso no hay confusión: si hubiera más hijos tendría que usar el identificador generado en su llamada) ha superado el valor 500 de su variable local ***Y***. Si es el caso, se le lanza una señal de matarlo, y seguidamente, se le vuelve a crear, volviéndose a ejecutar el código del proceso “player()”, el cual sitúa el jugador otra vez en las coordenadas X=600 e Y=330 para volver a empezar. Tal como está el juego ahora, haber escrito *signal(son,s_kill)* o *let_me_alone()* sería equivalente, porque sólo hay un proceso más aparte del principal.

Supongo que te habrás dado cuenta de un gran fallo: nuestro personaje sólo cae en el precipicio si viene de un salto, pero si va caminando, ¡pasa por encima de él como si nada!Esto hay que solucionarlo.Vuelve al mapa de durezas. Ahora vas a repintar el hueco del precipicio (que había pasado de ser de color rojo a negro, como todo el resto del mapa) de otro color, el azul puro, por ejemplo. Y en el bucle Loop/End del proceso “player()”, justo antes del Frame añadimos lo siguiente:

```
if(map_get_pixel(0,27,x,y)==rgb(0,0,255))
          REPEAT
                    y=y+5;
                    frame;
          UNTIL (y>=500)
End
```

Lo que hace esto es: si el personaje toca la zona azul del mapa de durezas, le aumentará el valor de su variable ***Y*** para hacerlo caer hasta que desaparezca por debajo de la ventana. Y ya que cuando ocurre eso, ya has añadido la orden de enviar la señal de matarle, ya lo tenemos todo. Fíjate que hemos puesto la orden Frame; también dentro de este If: esto es para que se visualize la caida lentamente; si no se hubiera puesto, el Repeat/Until se haría de golpe y veríamos nuestro personaje de repente ya abajo, efecto que no queda nada bien.

Si has hecho unas cuantas plataformas y has probado de hacer que nuestro personaje salte y llegue a alguna de ellas, verás que una vez haya salido de la plataforma por un extremo, el personaje continuará caminando como si la plataforma no hubiera acabado, ¡y no caerá! Bueno. El truco está en poner, en el mapa de durezas, en cada extremo de la plataforma un pequeño cuadrado de otro color, por ejemplo el verde puro, de manera que quede un rectángulo rojo acabado por ambos extremos por cuadrados verdes –como una varita mágica multicolor-. Y escribir lo siguiente en el bucle Loop/End del proceso “player()”, justo antes del Frame:

```
if(map_get_pixel(0,27,x,y)==rgb(0,255,0))
          REPEAT
                    y=y+5;
                    frame;
          UNTIL (map_get_pixel(0,27,x,y)==rgb(255,0,0))
End
```

La idea es repetir el truco que hemos hecho antes para que nuestro personaje cayera por el precipio del suelo, pero ahora, la condición de final de la caida no es que salga por debajo de la pantalla sino que su nuestro personaje se tope con una zona roja en el mapa de durezas, que puede ser el suelo u otra plataforma inferior. En ese momento, la caída finalizará.

Todavía falta un detalle más, y es ver qué es lo que tenemos que hacer para que cuando nuestro personaje está caminando por una plataforma inferior y salta, si choca de cabeza con la parte inferior de una plataforma más elevada rebote y caiga otra vez para abajo. Tal como tenemos ahora el juego, cuando choca contra una plataforma en mitad del salto, se mantiene en esa plataforma, y eso no debe ocurrir. ¿Qué tendríamos que hacer? Pues lo mismo de siempre: crear una zona de un color determinado en el mapa de durezas que abarcara toda la zona inferior de las plataformas flotantes, y utilizar **map_get_pixel()** para detectar si la parte superior del gráfico del personaje -es decir, un punto dado por la diferencia entre su coordenada *Y* actual y la mitad de la altura total del gráfico del personaje- colisiona con ésta. Si lo hace, el personaje debería de cae igual que lo hace cuando se lanza por un extremo de cualquier plataforma o por el precipicio. Se deja como ejercicio, aunque si no se te ocurre, échale un vistazo al código del final.

Lo que queda por añadir son los posibles premios y/o enemigos que puedan surgir a lo largo del camino de nuestro protagonista. Asociado a ellos también deberíamos contabilizar por ejemplo los puntos que pueda llegar a ganar nuestro protagonista si logra alcanzar los premios, o la energía que pueda llegar a perder si no sortea bien los obstáculos. Se podría poner un límite de puntos para alcanzar el fin del nivel, o por el contrario, se podría obligar a un Game Over si nuestro protagonista se queda sin energía. Todos estos detalles se parecen bastante a los que ya vimos con el tutorial del matamarcianos y, si te acuerdas de aquel tutorial, no te serán demasiado difíciles de programar.

La gran diferencia respecto el tutorial del matamarcianos de capítulos anteriores viene de que los premios y los enemigos no deben aparecer en pantalla tan aleatoriamente como en dicho tutorial. Ambos tendrán que estar siempre sobre alguna plataforma: al igual que nuestro protagonista, no pueden moverse por el aire, (al menos a priori).Lo único realmente aleatorio en los premios y enemigos es el intérvalo de sus apariciones, ya que sus posiciones, repito, siempre estarán ligadas a unas coordenadas ***X*** e ***Y*** concretas, las que corresponden a la anchura y altura de las plataformas existentes en ese momento en pantalla (sus posiciones dentro de los límites de éstas sí que podrán ser aleatorias,pero sólo dentro de dichos límites). Los enemigos, a diferencia de los premios, podemos hacer que se muevan -de forma aleatoria, o acercándose en dirección al persojane principal, o como queramos-, pero siempre respetando también los límites que imponen a sus coordenadas ***X*** e ***Y*** las plataformas existentes en el juego. No es excesivamente complejo llevar a cabo en el código fuente todo este conjunto de consideraciones, aunque tampoco es trivial. Se puede llegar de múltiples formas, y todas serán buenas si funcionan. Lo más normal es utilizar puntos de control del escenario para ubicar los premios y los enemigos, aunque en nuestro código de ejemplo no lo haremos así, por comodidad.

A continuación puedes ver una implementación de estas ideas en el siguiente código comentado. Este codigo corresponde al juego completo de plataformas que hemos ido desarrollando hasta ahora, incluyendo apariciones de premios y enemigos, y recuento de puntos y game over.Para incluir los gráficos de los premios en el FPG hemos hecho servir los códigos 009 y 010 -hay dos gráficos de premios posibles,pues-. Además, cuando el personaje choque contra un premio, éste, después de darle puntos, desaparecerá, no sin antes cambiar su gráfico por el de una bonito gráfico de colores vivos, identificado con el código 011 en el FPG. Para incluir los gráficos de los enemigos en el FPG hemos hecho servir los códigos 006,007 y 008 -hay tres gráficos de enemigos posibles, pues-. Además, cuando el personaje choque contra un enemigo, éste, después de quitarle vida, desaparecerá, no sin antes cambiar su gráfico por el de una horrible explosión formada por gráficos de identificadores 012, 013, 014, 015 y 016. También se ha añadido por pura estética una pequeña introducción, y necesitarás algún que otro archivo de sonido y de fuentes FNT. Si no los tienes a mano, comenta las líneas que hacen uso de sonido y asigna el valor de 0 a las variables *idfuente* y *idfuente2*.

```
import "mod_video";
import "mod_map";
import "mod_scroll";
import "mod_wm";
import "mod_sound";
import "mod_text";
import "mod_rand";
import "mod_proc";
import "mod_key";
import "mod_draw";
import "mod_screen";
import "mod_grproc";
import "mod_math";

global
```

```
           int idescenario;
           int idplayer;
           int idfuente;
           int idfuente2;
           int idbandaso;
           int idding;
           int idcrash;
           int idtrumpet;
           int puntos;
           int vida;
end

process main()
private
end
begin
           set_mode(640,480,32);
           set_fps(60,1);
           set_title("Mi juego de plataformas");
           idfuente2=load_fnt("fuentes/fuente2.fnt");
           idfuente=load_fnt("fuentes/fuente.fnt");
           idescenario=load_fpg("plat.fpg");
           idbandaso=load_song("sonidos/mkend.mid");
           idding=load_wav("sonidos/ding.wav");
           idcrash=load_wav("sonidos/crash.wav");
           idtrumpet=load_wav("sonidos/trumpet1.wav");
           intro();
end

process intro()
begin
           x=320;
           y=240;
           file=idescenario;
           graph=19;
           play_wav(idtrumpet,0);
           from z=0 to 49; frame;end
           fade(0,0,0,2);
           while(fading) frame;end
           fade_on();
           juego();
end

process juego()
begin
           write(idfuente,500,30,4,"Puntos:");
           write(idfuente,500,80,4,"Vida:");
           start_scroll(0,idescenario,1,0,0,1);
           play_song(idbandaso,-1);
           idplayer=player();
           loop
/*En cada frame tengo un 10% de probabilidad de crear un malo y un 30% de crear un
premio. Ambos tendrán una posición X e Y aleatoria que puede ser cualquiera  de dentro
del scroll, un gráfico aleatorio entre unos cuantos disponibles y un tamaño aleatorio
también -entre la mitad y el doble que el original-. En párrafos anteriores en este
capítulo se ha comentado que los malos y los premios sólo pueden  aparecer (a priori)
sobre las plataformas existentes en el juego, y aquí los estamos creando en cualquier
parte. Bueno, tal como veremos luego cuando veamos el código de los procesos "malo" y
"premio", es una vez creado el malo o premio, cuando se comprueba si su X e Y
corresponden a las coordenadas válidas. Si es así, se continúa ejecutando la instancia
del malo o premio correspondiente; si no, se mata la instancia.*/
              if(rand(1,100)<10)malo(rand(0,2000),rand(0,480),rand(6,8),rand(50,150));end
            if(rand(1,100)<30)premio(rand(0,2000),rand(0,480),rand(9,10),rand(50,150));end
/*If necesario para volver a comenzar otra vez desde el principio cuando el personaje
haya desaparecido por debajo de la pantalla al caer por un precipicio*/
              if(idplayer.y>=500)
                     //signal(idplayer,s_kill);
                     let_me_alone();
```

```
                    idplayer=player();
            end
            frame;
        end
end

process player()
private
        int xx=6;
        int xxx=12;
        int yy=20;
        int yyy=10;
        int iy=1;
        int cont=0;
        int obstacle=0;
        int avanzadillader;
        int avanzadillaizq;
        int idpuntos;
        int idvida;
end
begin
        ctype=c_scroll;
        scroll[0].camera=id;
        x=600;
        y=280;
        graph=3;
        puntos=0;
        vida=10;
        loop
                    idpuntos=write_var(idfuente,600,30,4,puntos);
                    idvida=write_var(idfuente,600,80,4,vida);
/*En cada fotograma actualizo los valores de estas dos variables, que servirán para hacer
un map_get_pixel de 20 píxeles más allá de donde está el personaje (avanzadillader si el
personaje va hacia la derecha y  viceversa) para comprovar que se choca contra un
obstaculo y por tanto se ha de parar. Para desatascarse, al cambiar el flag la
avanzadilla cambia y por tanto, en vez de comprobar el pixel de 20 más allá, se comprueba
el pixel de 20 más acá y el personaje se podrá desatascar*/
                    avanzadillader=x+20;
                    avanzadillaizq=x-20;
/*Separo este if del siguiente porque lo que me interesa es que cuando choque contra
algo, y por tanto, obstacle será 1, el flags se cambie igualmente. Y esto es importante
para los elseifs que vienen a continuación, donde dependiendo del valor de flags, se
utilizará avanzadillader o avanzadillaizq para poder desatascar el personaje*/
                    if(key(_right)) flags=0; end
/*Si se mueve hacia la derecha y no hay obstáculo, avanza, con la animación
correspondiente*/
                   if(key(_right) and obstacle==0)
                              x=x+10;
                               if(x>=2000) x=x-2000;end
                              for(cont=5;cont>2;cont=cont-1)
                                        graph=cont;
                                        frame;
                              end
                    end
                    /*Lo mismo que antes, para la izquierda*/
                    if(key(_left))flags=1; end
                    /*Lo mismo que antes, para la izquierda*/
                    if(key(_left) and obstacle==0)
                               x=x-10;
                              if(x<=0)x=x+2000;end
                              for(cont=5;cont>2;cont=cont-1)
                                        graph=cont;
                                        frame;
                              end
                    end

/*Bloque que comprueba si la parte superior de la cabeza del personaje (y-40), y  por
tanto, el personaje en sí, choca contra una plataforma que es más baja que él.Si choca,
```

```
se pone la variable obstacle a 1, y por tanto impedirá en los ifs anteriores que el
personaje se pueda mover. Notar el uso de las avanzadillas en vez de la x, para poder
desatascar el personaje, y también notar el uso de elseifs excluyentes entre  sí, por que
si hacen con ifs/else se pueden solapar unos con otros!!*/
    if(flags==0 and map_get_pixel(idescenario,2,avanzadillader,y-40)==rgb(255,0,0))
                                obstacle=1;
    elseif (flags==0 and map_get_pixel(idescenario,2,avanzadillader,y-40)!=rgb(255,0,0))
                                obstacle=0;
    elif(flags==1 and map_get_pixel(idescenario,2,avanzadillaizq,y-40)==rgb(255,0,0))
                                obstacle=1;
    elif (flags==1 and map_get_pixel(idescenario,2,avanzadillaizq,y-40)!=rgb(255,0,0))
                                obstacle=0;
                        end

                        /*Salto hacia la derecha*/
                        if(key(_s) and flags==0 and obstacle==0)
                              repeat
                                   /*El movimiento del salto propiamente dicho*/
                                   y=y-(yy-iy);
                                   x=x+xx;
                                   /*Para no salirse del mapa de durezas*/
                                   if(x>=2000) x=x-2000;end
/*Cuando la cabeza del personaje rebote con la base de una plataforma, éste ha de caer.
El algoritmo es el mismo que el de la caida libre desde una plataforma superior a otra
inferior (ver más abajo)*/
                                    if(map_get_pixel(idescenario,2,x,y-40)==rgb(255,255,0))
                                     while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                                    y=y+5;
                                                    frame;
                                     end
                                     break;
                                     end
                                     frame;
//Para que se vaya desacelerando el salto y el personaje vuelva a bajar
                                     iy=iy+1;
//Falta hacer que mientras esté saltando y choque contra una plataforma,se detenga el salto
                              until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
/*or map_get_pixel(idescenario,2,avanzadillader,y)==rgb(255,0,0))*/
                              iy=1;
                        end

/*Salto hacia la izquierda. Igual que el de la derecha excepto en que la coordenada x
disminuye en vez de aumentar.Por lo demás, todo igual.*/
                        if(key(_s) and flags==1 and obstacle==0)
                              repeat
                                   y=y-(yy-iy);
                                   x=x-xx;
                                   if(x<=0) x=x+2000;end
                                   if(map_get_pixel(idescenario,2,x,y-40)==rgb(255,255,0))
                                    while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                                    y=y+5;
                                                    frame;
                                   end
                                   break;
                                   end
                                   frame;
                                   iy=iy+1;
                              until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
                              iy=1;
                        end

                        /*Salto hacia la derecha*/
                        if(key(_d) and flags==0 and obstacle==0)
                              repeat
                                   /*El movimiento del salto propiamente dicho*/
                                   y=y-(yyy-iy);
                                   x=x+xxx;
                                   /*Para no salirse del mapa de durezas*/
```

```
                                if(x>=2000) x=x-2000;end
/*Cuando la cabeza del personaje rebote con la base de una plataforma, éste ha de caer.
El algoritmo es el mismo que el de la caida libre desde una plataforma superior a otra
inferior (ver más abajo)*/
                                if(map_get_pixel(idescenario,2,x,y-40)==rgb(255,255,0))
                                 while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                          y=y+5;
                                          frame;
                                 end
                                 break;
                                end
                                frame;
          //Para que se vaya desacelerando el salto y el personaje vuelva a bajar
                                iy=iy+1;
//Falta hacer que mientras esté saltando y choque contra una plataforma, se detenga el salto
                               until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
/*or map_get_pixel(idescenario,2,avanzadillader,y)==rgb(255,0,0))*/
                               iy=1;
                         end

/*Salto hacia la izquierda. Igual que el de la derecha excepto en que la coordenada x
disminuye en vez de aumentar.Por lo demás, todo igual.*/
                      if(key(_d) and flags==1 and obstacle==0)
                            repeat
                                y=y-(yyy-iy);
                                x=x-xxx;
                                if(x<=0) x=x+2000;end
                                if(map_get_pixel(idescenario,2,x,y-40)==rgb(255,255,0))
                                 while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                          y=y+5;
                                          frame;
                                 end
                                 break;
                                end
                                frame;
                                iy=iy+1;
                               until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
                            iy=1;
                      end


/*Cuando se encuentra un precipicio, el personaje cae y desaparece -y muere-*/
                      if(map_get_pixel(idescenario,2,x,y+45)==rgb(0,0,255))
                                while(y<500)
                                          y=y+5;
                                          frame;
                                end
                      end

/*Cuando el personaje se encuentra en una plataforma y ésta se acaba, ha de caer hasta la
plataforma de debajo*/
                      if(map_get_pixel(idescenario,2,x,y+45)==rgb(0,255,0))
                                while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                          y=y+5;
                                          frame;
                                end
                      end

                      if(puntos==30)
                                 write(idfuente2,320,280,4,"¡MUY BIEN!");
                                 write(idfuente2,320,320,4,"Pulse ESC para salir");
                                 stop_scroll(0);
                                 put_screen(idescenario,17);
                                 while (not key(_esc))
                                                    frame;
                                 end
                                 exit();
                      end
```

```
                    if(vida==0)
                              write(idfuente,320,240,4,"GAME OVER!");
                              write(idfuente,320,280,4,"Pulse ESC para salir");
                              stop_scroll(0);
                              put_screen(idescenario,18);
                              while (not key(_esc))
                                                  frame;
                              end
                              exit();
                    end

                    frame;
                    delete_text(idpuntos);
                    delete_text(idvida);
          end
end

process malo(x,y,graph,size)
private
          int cont;
          int angleplay;
          int distplay;
          int move_x;
          int yasalio=0;
end
begin
ctype=c_scroll;
repeat
/*Cada paréntesis que hay entre los "or" indica la posición del suelo superior de cada
una de las diferentes plataformas pintadas en nuestro fondo de scroll. Se comprueba que
la posición del malo recién creado (tanto horizontal como verticla) esté en alguna zona
del scroll correspondiente al ras de suelo  de alguna de las plataformas existentes. Esto
se hace mirando los valores de los extremos horizontales izquierdo y derecho y una franja
estrecha alrededor del extremo vertical superior de las plataformas. Esta línea de código
la tendrás que cambiar para que se adecúe al dibujo que hayas hecho tu del mapa de
plataformas. No se ha contemplado la posibilidad de que puedan existir enemigos fuera de
estas zonas (volando en el cielo, por ejemplo).*/
          if((x>0 and x<530 and y>410 and y<430)or(x>125 and x<375 and y>165 and y<185)
or (x>465 and x<715 and y>290 and y<310) or (x>1280 and x<1500 and y>290 and y<310) or
(x>1470 and x<1710 and y>210 and y<230) or (x>1740 and x<1970 and y>390 and y<410) or
(x>625 and x<1315 and y>415 and y<435))
/*Las siguientes líneas sirven para detectar la presencia cercana del personaje, en cuyo
caso  los enemigos irían tras él. Son exactemente las mismas líneas utilizadas en el
tutorial del RPG, en un capítulo anterior. Lo que faltaria por hacer es que los enemigos
puedan moverse entre plataformas,por ejemplo.*/
               angleplay=get_angle(idplayer);
               distplay=get_dist(idplayer);
               move_x=get_distx(angleplay,distplay);
               if(move_x<-20 and move_x>-100) x=x-1;yasalio=1; end
               if(move_x>20 and move_x<100) x=x+1; yasalio=1;end
                    frame;
               else
                    if(yasalio==0)
                         signal(id,s_kill);
/*Importante este else, porque si se da el caso de que ya ha salido y se ha llegado al
extremo de una plataforma (es decir,cuando se haga este else, porque es cuando yasalio=1
y cuando la x y la y son diferentes de las del primer if, que son los que indican las
posiciones de las plataformas), el programa se quedaría colgado al no haber un frame;*/
                    else
                         frame;
                    end
                end
until(collision (idplayer))
/*Con sólo un choque contra al jugador, el malo muere, pero decrece en una unidad la vida
del personaje,  indistintamente del tipo de gráfico de malo que sea (una aspecto para
mejorar...)*/
play_wav(idcrash,0);
```

```
vida=vida-1;
cont=12;
for(cont=12;cont<=16;cont=cont+1)
          graph=cont;
          frame;
end
end

process premio(x,y,graph,size)
private
          int cont;
end
begin
ctype=c_scroll;
repeat
/*La condición de este if es idéntica a la de los malos, ya que sólo podrán aparecer
sobre las plataformas existentes, al igual que ellos*/
          if((x>0 and x<530 and y>410 and y<430)or(x>125 and x<375 and y>165 and y<185)
or (x>465 and x<715 and y>290 and y<310) or (x>1280 and x<1500 and y>290 and y<310) or
(x>1470 and x<1710 and y>210 and y<230) or (x>1740 and x<1970 and y>390 and y<410) or
(x>625 and x<1315 and y>415 and y<435))
                    frame;
          else
                    signal(id,s_kill);
           end
until(collision (idplayer))
play_wav(idding,0);
/*Con sólo un choque contra al jugador, el premio desaparece, pero aumenta en una unidad
los puntos del personaje, indistintamente del tipo de gráfico de premio que sea (una
aspecto para mejorar...).*/
puntos=puntos+1;
graph=11;
for(cont=1;cont<=50;cont=cont+1)
          y=y-10;
          frame;
end
end
```

Ya tenemos acabado nuestro juego de plataformas básico, pero es posible que quieras mejorarlo. Una de las posibles mejoras podría ser por ejemplo el introducir más niveles, los cuales incorporarían otras plataformas diferentes. A lo mejor te puede apetecer crear plataformas de forma irregular, con pequeñas cuestas o bajadas. Si es éste el caso, el de incorporar pendientes a nuestro juego, tendrás que utilizar sabiamente el mapa de durezas para hacer que el personaje baje de forma correcta por una bajada y suba de forma correcta por una subida: no es tan fácil como en el juego precedente, pero bueno, es tan simple como utilizar 2 colores para crear el "suelo" (verde #00ff00 y amarillo #ffff00, por poner un ejemplo). El primer color sería la línea sobre la que camina el personaje, y evitaría que este caiga hacia abajo . Tendrías que usar un if más o menos así para ello:

```
if (color != rgb_verde)
    y=y+caida;
end
```

El segundo color, situado por debajo del primero, "empujaría" al personaje hacia arriba hasta que deje de tocar ese color (dejándolo justo sobre la línea verde, que es por la que camina el personaje). Tendrías que usar un while más o menos así para ello:

```
while (color == rgb_amarillo)
    y--;
end
```

Con esto, subiría cuestas de forma impecable.... y bajarlas, bueno, dependería de cómo apliques la gravedad y sobre todo, los cambios de gráfico... podría "caerse" constantemente al bajar una cuesta, o podría quedar también impecable: es cuestión de planearlo bien, nada más.

Por cierto, si te estás preguntando cómo añadir varios niveles a un juego de plataformas...¿te acuerdas del truco que utilizamos en el tutorial del RPG para pasar del mapa general al interior de una casa o al inventario? Exacto:

con un SWITCH que iba controlando en qué nivel estábamos en cada momento y ejecutaba el bucle correspondiente hasta que salíamos de éste. Pues con los niveles de los juegos de plataformas podemos utilizar el mismo sistema. Ésa sería una manera fácil.

Otro consejo, para posibles futuros juegos de plataformas que te animes a hacer:uno de las fallos que muchos programadores tienen a la hora de hacer un juego tipo plataformas es la cantidad de memoria y procesador que requieren éstos. Esto se debe a que una vez que se crea una fase, si esta tiene 800 monedas por ejemplo y 500 enemigos, todos cargados a la vez, aunque no esten en pantalla, estan gastando furiosamente memoria. El proceso que se presenta a continuación, "pon_monedas()" evita este problema, haciendo que a medida que el personaje se vaya moviendo, las monedas y demás objetos van apareciendo, es decir: se crean un poco antes de que vayan a aparecer, y cuando los dejamos atras, mueren. Ademas, si nuestro personaje eliminara alguno de esos procesos, ya sea recogiendo monedas o matando a los enemigos, dicho proceso ya no aparecera mas (como debe ser, claro). No obstante, antes de ver el código de dicho proceso hay que saber lo siguiente:

*Las monedas (o bichos, u objetos, o lo que sea) a crear ya no se crearan como tradicionalmente: simplemente hay que llamar una sola vez al proceso "pon_monedas()".

*En la sección GLOBAL del programa hay que definir dos matrices:

A) viva[Nro de monedas]; (Para indicar si cada moneda esta viva 1 o muerta 0)

B) id_coin[Nro de monedas]; (Identificador de cada moneda)

donde "Nro de monedas" es el numero de monedas maximo que pueda aparecer en cualquier fase del juego

*En la sección LOCAL del programa hay que definir la siguiente variable:

Numero; (Que sera el numero de cada moneda)

Ahora pasamos a ver el proceso suponiendo que el juego esta a una resolucion de 320x240, las monedas han sido llamadas "coin()", el personaje que las debe recoger es "personaje()", y tenemos por ejemplo, 200 monedas como maximo.

```
process pon_monedas()     // Puede ser "pon_monedas", "pon_bichos"... :)
private
       int numero_de_moneda[200]; // Una tabla que indica si cada moneda esta puesta o no
       int contador;              // Contador de las monedas a crear-matar-modificar
end
begin
from contador=1 TO 200;           // Este bucle crea TODAS las monedas
//Obtiene las coord de los puntos de control de un supuesto mapa nro 1
    get_point(0,1,contador,&x,&y);
    if (x<>0 AND y<>0) //Cuando las coordenadas son 0 quiere decir que no hay mas monedas
// Crea la moneda con las coords y su nº correspondiente
        id_coin[contador]=coin(x,y,contador);
        numero_de_moneda[contador]=1;      //Se indica que dicha moneda esta puesta (1)
        viva[contador]=1;                  //Se indica que dicha moneda esta viva (1)
    end
end
loop                     // Este bucle permanecera durante toda la fase actual del juego
    from contador=1 to 200;             // Este bucle estudiara cada moneda
//Obtiene las coords de cada moneda(los pntos de control del mapa)
        get_point(1,1,contador,&x,&y);

/* A continuacion se hacen dos comprobaciones, la primera consiste en averiguar si una
moneda que no esta en pantalla y que sigue viva, esta lista para ser puesta, y en la
segunda comprobacion se averigua si una moneda que esta viva y puesta debe ser quitada de
pantalla para ahorrar memoria.*/

/* 1ro Si en las coordenadas hay o habia moneda y se encuentra cerca del personaje y no
esta puesta en escena, pero esta viva*/
        if (x<>0 AND y<>0 AND x<id_personaje.x+320 AND x>id_personaje.x-320 AND
y<id_personaje.y+240  AND  y>id_personaje.y-240  AND  numero_de_moneda[contador]==0    AND
viva[contador]==1)
            // Creamos dicha moneda
            id_coin[contador]=coin(x,y,contador);
            // Y ahora indicamos que esta puesta
            numero_de_moneda[contador]=1;
        end

/* 2do Si en las coordenadas hay o habia moneda y se encuentra lejos del personaje y esta
```

```
puesta en escena y viva*/
       if  (x<>0  AND  y<>0  AND  (x>id_personaje.x+320  OR  x<id_personaje.x-320  OR
y>id_personaje.y+240  OR  y<id_personaje.y-240)  AND  numero_de_moneda[contador]==1  AND
viva[contador]==1)
            // Quitamos la moneda para ahorrar memoria
            signal(id_coin[contador],s_kill);
            // E indicamos que no esta puesta (aunque si viva)
            numero_de_moneda[contador]=0;
       end
    end //From
    frame;
end //Loop
end
```

Un detalle interesante del código anterior es que para situar las monedas (en este caso), en vez de utilizar el método bastante rudimentario que utilizamos en el tutorial anterior de depender de las posiciones de las plataformas, aquí se utiliza otro método más profesional: puntos del control dentro del mapa de scroll. Es decir, las monedas no las crearemos poniendo nosotros las coordenadas, sino colocando puntos de control en el mapa donde aparecerán dichas monedas. El proceso se encargara de quitarlas o ponerlas cuando le corresponda.

Ademas de esto, hay que tener en cuenta que cuando el personaje toque una moneda, ésta de morir definitivamente, cambiando su variable *viva* a 0. Deberíamos incluir en el proceso del personaje algo como esto:

```
IF (id2=collision(TYPE coin))   // Si toca una moneda
        viva[id2.numero]=0;     // le quita la vida
        signal(id2,s_kill);     // y la quita de pantalla
END
```

En tu afán por mejorar y ampliar tu juego de plataformas, se te puede ocurrir introducir plataformas móviles. Esto complica considerablemente la programación, pero ofrece un acabado del juego muy superior al estándar. Básicamente, si quieres implementar un sistema de plataformas móviles, lo que tendrías que hacer es que el bucle del personaje principal fuera similar a lo siguiente (hablando abstractamente):

```
/*'avance' es una variable local que se refiere a la dirección hacia la que el personaje
mira (-1 para izquierda y +1 para la derecha), 'inc_x' es una variable privada que se
refiere al incremento horizontal oportuno en el movimiento del personaje(según esté
agachado, levantado, saltando) en valor absoluto.'impulso' es una variable privada que
cuanto más grande sea más impulso vertical tendrá el protagonista, y por tanto, más
arriba. 'velocidad' es una variable local que indica la velocidad de desplazamiento de
una plataforma 'en_plataforma' es una variable privada al personaje que adopta valores
'si' o 'no' (1 ó 0) e indica si estoy en una plataforma o no 'id2' lo uso para recojer
las colisiones con plataformas*/

Loop
            // ACTUALIZAR x
          If(en_plataforma) // ¿Estoy en una plataforma?
                    x=x+id2.velocidad;
          End
           // ahora actualizo: le sumo a x el incremento en la dirección adecuada
          x=x+inc_x*avance;

          // ACTUALIZAR y (algoritmo típico de salto)
          If(impulso<>0) y=y-impulso;impulso--; End

          // Calcula la gravedad, mi gravedad va aparte, más adelante te la enseño.
          id_gravedad=gravity();

          // COLISION CONTRA UNA PLATAFORMA
          id2=collision(Type plataforma); // Cojo el ID de la plataforma
          If(id2)
                    // Si esta sobre ella y callendo
                    If(y<id2.y+2)
                              en_plataforma=si; saltando=no;
                    Else
                             en_plataforma=no;
                    End
          Else
```

```
                    en_plataforma=no;
          End

/* Se mantiene junto a la plataforma, ajusto automáticamente la 'y' del personaje en
plataformas verticales*/
          If(en_plataforma==si) y=id2.y; End
          Frame; // Refresco
End
```

Y el resto de procesos necesarios...:

```
Process gravity()
Begin
ctype=c_scroll;

/* Si el proceso afectado por la gravedad se encuentra en una plataforma, queda afectado
por otros factores determinados  en su propio bucle principal. En caso contrario... */
If(father.en_plataforma<>1)
          // El color 255 (rojo) es el suelo
          If(suelo(father.x,father.y)<>255)
                    father.saltando=1;
          Else
                    father.saltando=0;
                    gravedad=1;
                    End

          // La caida se controla punto a punto
          If(father.saltando==1)
                    For (yy=father.y; yy<=father.y+gravedad; yy++)
                              If(suelo(father.x,yy)==11) father.y=yy; Break; End
                    End
          End

          // El límite de la gravedad es 75
          If(father.saltando==1)
                    If(gravedad < 75); gravedad+=8.5; Else; gravedad=75;End
                    // Se actualiza la posicion del objeto afectado
                    father.y+=gravedad;
          End
End
End

// El desplazamiento sigue el movimiento armónico simple (como el del muelle)
Process plataforma(file,graph,x,y,tipo,desp,velocidad)
Begin
ctype=c_scroll;
z=id_personaje.z+1;
cx=x;
cy=y;

// Plataforma Horizontal
If(tipo==0)xx=x-desp; yy=x+desp;End

// Plataforma Vertical
If(tipo==1) xx=y-desp; yy=y+desp; End

Loop
          // Desplazamiento horizontal
          If(tipo==0)
                    If(x<xx OR x>yy) velocidad=-velocidad; End
                    x+=velocidad;
          End

          // Desplazamiento vertical
          if(tipo==1)
                    If(y<xx OR y>yy) velocidad=-velocidad; End
                    y+=velocidad;
          End
```

```
            Frame;
End
End
```

## *Códigos gestores de FPGs

En el capítulo 3 se introdujeron varios códigos de ejemplo de programas Bennu que eran capaces de crear ficheros FPG y añadir ó eliminar imágenes dentro de éstos. No obstante, debido a su elevada dificultad no se incluyeron un par de códigos con estos mismos objetivos, para no desorientar al lector más de lo necesario. No obstante, se ha considerado incluir dichos interesantes códigos de ejemplo en el apartado actual, para posibilitar su estudio a quien quiera hacerlo.

```
/*A partir de este código se podría hacer muy fácilmente una función, con dos parámetros:
la carpeta donde estarían los PNG y el número máximo de PNGs a cargar y que retornaría el
identificador del FPG creado (ó códigos de error). Lo interesante de este ejemplo es que
se ordenan alfabéticamente los nombres de las imágenes PNG para incluirlas en el FPG*/
import "mod_map";
import "mod_dir";
import "mod_mem";
import "mod_string";
import "mod_sort";
import "mod_proc";
import "mod_text";
import "mod_key";

Process Main()
Private
    int i;
    int f;
    int g;
    String filename;
    String pointer entry=null;
    int entries = 0;
    String oldfolder;
    int maxnumber;
End
Begin
// Recordar la carpeta actual (para volver a ella al final del programa)
    oldfolder = cd();
/*Mirar si el nombre de la carpeta es correcto, y si lo es, se mueve a su interior.Si no,
aborta con mensaje de error.Este nombre se podría haber introducido como parámetro del
programa, o de forma interactiva...esto habría que implementarlo.*/
    if(chdir(".")!=0) exit("Error:carpeta no válida",-1); end
/*Otro parámetro que se le puede pasar al programa es el número máximo de PNGs que se
desean introducir, teniendo en cuenta por seguridad además que si se introduciera un
valor negativo, se establecería automáticamente a 999. En este ejemplo, estableceremos
maxnumber a 100. Ojo,porque si se establece un número máximo menor del número de PNGs
existentes en la carpeta de trabajo, el programa se cuelga. */
    maxnumber=100; if(maxnumber<0) maxnumber = 999;  end
/*Recoger todos los ficheros PNG de la carpeta especificada. Se utiliza para ello una
matriz de cadenas, donde se almacenarán los diferentes nombres de los ficheros PNG, cuyo
tamaño viene dado por el número máximo de elementos que se desea incluir*/
    entry = alloc(maxnumber*sizeof(string));
    while(len(filename=glob("*.png"))>0)
        //Si es un directorio, no hacemos nada y saltamos al siguiente
        if(fileinfo.directory) continue; end
        entry[entries] = filename;
        entries++;
    end
//Ordenar alfabéticamente la lista de ficheros PNG
    sort(entry,entries);
//Cargar estos ficheros ya ordenados en un nuevo contenedor FPG
    f = fpg_new();
    for(i=0; i<entries; i++)
```

```
        g = load_png(entry[i]);
        if(g<=0) exit("Error cargando imagen",-2); end
        fpg_add(f,i,0,g);
        //Establecemos el nombre de la imagen como descripción dentro del FPG
        map_set_name(f,i,entry[i]);
        unload_map(0,g);
    end
/*Una vez ya tenemos el FPG creado, podemos hacer lo que queramos con él. Por ejemplo,
podemos comprobar que efectivamente se haya creado bien con las 100 imágenes PNG (como
máximo) en su interior.*/
    for(i=0; i<100; i++)
  write(0,0,i*10,0,i+":"+map_name(f,i)+"("+(map_exists(f,i)?"OK":"FAILED")+")");
    end
    while(!key(_esc)) frame; end
//Se libera el espacio usado por la lista ordenada de nombres de ficheros PNG
    if(entry) free(entry); end
//Se vuelve a la carpeta original
    chdir(oldfolder);
End
```

Otro ejemplo:

```
/*Código que genera un FPG diferente para cada subcarpeta dentro de la carpeta actual,
con los archivos PNG que respectivamente hay en su interior. Estos archivos PNG se han de
llamar obligatoriamente con un número: 1.png, 23.png, 536.png, etc*/
Import "mod_map";
Import "mod_key";
Import "mod_string";
Import "mod_file";
Import "mod_dir";
Import "mod_video";

process main()
private
        string carpeta;
        int idfpg,idgraf;
        int i;
end
Begin
          set_mode(100,100,32);
/*Mientras se vaya listando el contenido de la carpeta actual (ficheros y carpetas uno a
uno...)*/
          While((carpeta=glob("*")) != "")
//Si la carpeta es la carpeta actual o la anterior, no se hace nada
                    if(carpeta!=".." or carpeta!=".")
                              cd(carpeta);
                              idfpg=fpg_new();
                              from i=0 to 999;
                                        If(file_exists(itoa(i)+".png"))
                                                  idgraf=load_png(itoa(i)+".png");
                                                  fpg_add(idfpg,i,0,idgraf);
//Después de añadir al fpg la imagen,la descargo para continuar con la siguiente
                                                  unload_map(idfpg,idgraf);
                                        end
                              End
                              save_fpg(idfpg,"../"+carpeta+".fpg");
/*Ya he creado el fpg de una carpeta determinada. Ahora descargo su identificador para
reutilizarlo en el siguiente fpg que se creará de la próxima carpeta */
                              unload_fpg(idfpg);
                              cd("..");
                    end
          End
End
```

## *Códigos de un picaladrillos, un ping-pong y un matamarcianos sin utilizar ninguna imagen externa

A continuación presento un código comentado de un picaladrillos, otro de un ping-pong y otro de un matamarcianos, los cuales tienen en común que todos los gráficos de todos sus procesos son generados internamente en el código fuente, con lo que no necesitaremos ningún fichero externo para poder ejecutar el juego.

Estos códigos nos pueden servir como demostración rápida al público advenedizo de lo que se puede llegar a hacer con Bennu en una pocas líneas, de forma sencilla y rápida, a modo de "prueba de concepto".

Empezamos por el picaladrillos:

```
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_key";
import "mod_video";
import "mod_grproc";
import "mod_draw";
import "mod_rand";
import "mod_screen";

global
          int XSize=200,YSize=400;
          int xIncrement=5,yIncrement=-5;
end

process main()
private
          int n,a;
end
begin
          set_mode(XSize,YSize,32);
          Barra();
          Bola();
/*Dibujo los ladrillos. Se pintarán 19 filas y, si la anchura de la pantalla es 200, se
pintarán 7 ladrillos por columna */
          from n=10 to XSize-10 step 30;
                    from a=20 to 200 step 10;
                              Bloque(n,a);
                    end
          end
           repeat frame; until (key(_esc))
          exit();
end

PROCESS Barra()
BEGIN
          graph=new_map(50,10,32);map_clear(0,graph,rgb(100,100,100));
/*La posición inicial de la raqueta dependerá de las dimensiones de la pantalla. En todo
caso, aparecerá centrada horizontalmente  y cerca de su límite inferior*/
          y=YSize-40;
          x=XSize/2;
          LOOP
                    if (key(_left))
                              if (x>0) x=x-5; end
                    elif (key(_right))
                              if (x<XSize) x+=5; end
                    end
/*Si hay colisión con la bola, invierto el incremento del movimiento de ésta en la
dirección vertical*/
                    if (Collision(type Bola)) yIncrement=-yIncrement; end
                    frame;
          END
END

PROCESS Bola()
```

```
BEGIN
          graph=new_map(10,10,32);
          drawing_map(0,graph);
          drawing_color(rgb(255,0,0));
          draw_fcircle(5,5,5);

/*La posición inicial de la raqueta dependerá de las dimensiones de la pantalla. En todo
caso, aparecerá centrada horizontalmente  y cerca de su límite inferior, por encima de la
raqueta.*/
          x=XSize/2;y=ySize-50;
          LOOP
                    if (x>=XSize OR x<=0)
          //Invierto el incremento del movimiento de la bola en la dirección horizontal
                              xIncrement=-xIncrement;
                    elseif (y<=0)
                              yIncrement=-yIncrement;
                    end

/*Si la bola desaparece por el límite inferior de la pantalla, se acabó el juego, ya que
(después de ejecutar el proceso "Perdedor", entro en un bucle infinito sin salida,  con
lo que la bola deja de ser funcional.*/
                    if (y>=YSize)
                              Perdedor();
                              loop frame; end
                    end

                    //Movimiento de la bola
                    x=x+xIncrement;
                    y=y+yIncrement;

                    frame;
          END
END

PROCESS Bloque(INT x,y)
BEGIN
          z=100;
/*Los ladrillos serán de 30x10. Su tamaño es fijo, pero el número de ellos dependerá de
las dimensiones de la pantalla*/
          graph=new_map(30,10,32);
          map_clear(0,graph,rgb(rand(0,255),rand(0,255),rand(0,255)));

          LOOP
/*Si hay colisión con la bola, invierto el incremento del movimiento de ésta en la
dirección vertical, y termino inmediatemente la ejecución de la instancia actual, con lo
que el ladrillo desaparecerá de la pantalla.*/
                    if (Collision(TYPE Bola))
                              yIncrement=-yIncrement;
                              return;
                    end
                    frame;
          END
END

/*Escribo 200 veces en posiciones aleatorias y con colores y profundidades aleatorias
también el texto (convertido en imagen)"LOOOSERRRR"*/
PROCESS Perdedor()
PRIVATE
          INT aux;
END
BEGIN
          FROM x=1 TO 200;
                    text_z=-400;
                    set_text_color(rgb(rand(0,255),rand(0,255),rand(0,255)));
                    aux=write_in_map(0,"LOOSERRRR",0);
                    put(0,aux,rand(0,XSize),rand(0,YSize));
                    unload_map(0,aux);
          END
```

```
END
```

A continuación, el ping-pong. La lógica del juego todavía no está acabada (la pelota no hace más que rebotar horizontalmente) pero lo interesante no es el juego en sí sino el acabado gráfico resultante, obtenido en tiempo real de ejecución del juego. Como valor añadido además, el código está escrito de tal manera que es totalmente escalable, ya que todos los gráficos son proporcionales a la resolución señalada en cada momento: así, si se indica una resolución más pequeña, de forma automática las palas, la bola, los marcadores, etc amoldarán su nueva posición acordemente, y si la resolución es mayor, igual.

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_screen";
Import "mod_key";
Import "mod_grproc";
Import "mod_proc";
Import "mod_text";

Const
  blanco=15;
  azul=55;
  rojo=24;
  gris=6;
  grisoscuro=3;
  res_x=800;
  res_y=640;
End
Global
          int altura_marcadores;
          int filas;
          int columnas;
          int tamanocelda; //20 en 800x600
           int efepeese=60;
           int turno; //1: jugador izquierda, 2: jugador derecha
           int graf[5];
           int velocidad=3; //Velocidad de la bola
End
Process Main()
Begin
if(res_x>400) set_mode(res_x,res_y,16); else set_mode(res_x,res_y,16,mode_2xscale); end
set_fps(efepeese,9);
tamanocelda=res_x*0.025; //800 = 20
altura_marcadores=res_y/6;
filas=(res_y-altura_marcadores)/tamanocelda;
columnas=res_x/tamanocelda;
mouse.graph=graf[3];
hazgraficos();
pintafondo();
bola();
palo1(res_x/10,res_y/2, graf[0]);
palo2(9*res_x/10,res_y/2, graf[1]);
senalaturno();
End

Process hazgraficos()
Begin
//palo1
graf[0]=new_map(tamanocelda,tamanocelda*4,8);
drawing_map(0,graf[0]);
drawing_color(rojo);
draw_box(0,0,tamanocelda,tamanocelda*4);
drawing_color(rojo-2);
draw_rect(0,0,tamanocelda,tamanocelda*4);
//palo2
graf[1]=new_map(tamanocelda,tamanocelda*4,8);
drawing_map(0,graf[1]);
drawing_color(azul);
```

```
draw_box(0,0,tamanocelda,tamanocelda*4);
drawing_color(azul-2);
draw_rect(0,0,tamanocelda,tamanocelda*4);
//bola
graf[2]=new_map(tamanocelda*1.5,tamanocelda*1.5,8);
drawing_map(0,graf[2]);
drawing_color(blanco);
draw_fcircle(tamanocelda/1.5,tamanocelda/1.5,tamanocelda/1.5);
drawing_color(gris);
draw_circle(tamanocelda/1.5,tamanocelda/1.5,tamanocelda/1.5);
//raton
graf[3]=new_map(tamanocelda/2,tamanocelda/2,8);
drawing_map(0,graf[3]);
drawing_color(blanco);
draw_box(0,0,1,1);
draw_box(0,0,tamanocelda/2,tamanocelda/6);
draw_box(0,0,tamanocelda/6,tamanocelda/2);
draw_box(0,0,tamanocelda/4,tamanocelda/4);
End


Process pintafondo()
Private
           int i;
End
Begin
graph=new_map(res_x,res_y,8);
drawing_map(0,graph);
drawing_color(gris);
// caja marcadores
draw_box(0,0,res_x,altura_marcadores);
drawing_color(grisoscuro);
draw_rect(0,0,res_x,altura_marcadores);
// celda fondo
drawing_color(grisoscuro);
write(0,res_x/4,altura_marcadores - 10,4,"Marcador 1");
write(0,3*res_x/4,altura_marcadores - 10,4,"Marcador 2");
//filas
from i=0 to filas;
           draw_line(0,altura_marcadores+(i*tamanocelda),res_x,altura_marcadores+
(i*tamanocelda));
end
//columnas
from i=0 to columnas;
           draw_line(tamanocelda*i,altura_marcadores+1,tamanocelda*i,res_y);
end
//linea vertical del centro
drawing_color(gris);
draw_box((res_x/2)-2,altura_marcadores,(res_x/2)+2,res_y);
//linea roja vertical
drawing_color(rojo);
draw_box(0,altura_marcadores,res_x,altura_marcadores+1);
//linea azul vertical
drawing_color(azul);
draw_box(0,res_y-2,res_x,res_y);
//y lo ponemos de fondo
put_screen(0,graph);
End

Process bola()
private
           int ancho;
           int alto;
end
Begin
graph=graf[2];
x=res_x/2;
y=((res_y-altura_marcadores)/2)+altura_marcadores;
```

```
ancho=map_info(file,graph,g_width);
alto=map_info(file,graph,g_height);
loop
  if(x<res_x/2) turno=1;end
  if(x>res_x/2) turno=2;end
  if(collision( type palo1) or collision (type palo2))
                    angle=(angle+180000)%360000;
                    advance(velocidad);
  end
  advance(velocidad);
  sombra();
  frame;
end
End

Process palo1(x,y,graph)
private
          int ancho;
          int alto;
end
Begin
          ancho=map_info(file,graph,g_width);
          alto=map_info(file,graph,g_height);
          loop
                              if(key(_down)) y=y+2; end
                              if(key(_up)) y=y-2;end
                              sombra();
                    frame;
          end
End

Process palo2(x,y,graph)
private
          int ancho;
          int alto;
end
Begin
          ancho=map_info(file,graph,g_width);
          alto=map_info(file,graph,g_height);
          loop
                              if(key(_a)) y=y+2; end
                              if(key(_q)) y=y-2;end
                              sombra();
                    frame;
          end
End

Process sombra()
Begin
if(exists(father))
          file=father.file;
          graph=father.graph;
                    size=father.size;
                    x=father.x;
                    y=father.y;
                    z=father.z+1;
                  from alpha=128 to 0 step -8;
                frame;
                  end
else
          return;
end
End

Process senalaturno()
Private
          int color1=rojo;
          int color2=azul;
```

```
        int mapa1;
        int mapa2;
End
Begin
        mapa1=new_map(res_x/2,res_y-altura_marcadores,8);
        mapa2=new_map(res_x/2,res_y-altura_marcadores,8);
        y=((res_y-altura_marcadores)/2)+altura_marcadores;
        z=2;
        // mapa1
        drawing_map(0,mapa1);
        drawing_color(color1);
        draw_box(0,0,res_x/2,res_y-altura_marcadores);
        // mapa2
        drawing_map(0,mapa2);
        drawing_color(color2);
        draw_box(0,0,res_x/2,res_y-altura_marcadores);
        loop
                if(turno==1)
                        x=res_x/4;
                        graph=mapa1;
                        from alpha=0 to 64 step 4;
                                frame;
                        end
                        while(turno==1) frame; end
                        from alpha=64 to 0 step -6;
                                frame;
                        end
                end
                if(turno==2)
                        x=(res_x/4)*3;
                        graph=mapa2;
                        from alpha=0 to 64 step 4;
                                        frame;
                        end
                        while(turno==2) frame; end
                        from alpha=64 to 0 step -6;
                                        frame;
                        end
                end
                frame;
        end
End
```

Finalmente, el matamarcianos:

```
import "mod_map";
import "mod_text";
import "mod_proc";
import "mod_key";
import "mod_video";
import "mod_grproc";
import "mod_draw";
import "mod_rand";
import "mod_screen";
import "mod_wm";

global
    int tiro=1;//Si vale 1, el jugador puede usar el disparo especial.Si vale 0,no.
    int score=0;
end

process main()
begin
    set_fps(30,0);
    set_title("Marcianitos");
    set_mode (640,480,32);
    nave(400,500);
/*Inicialmente salen cinco enemigos.En el código posterior se puede ver que cuando un
```

```
enemigo desaparece de la pantalla a causa del impacto de un misil de nuestra nave,
inmediatamente se genera un nuevo enemigo en la parte superior de la pantalla -y con una
x aleatoria-. Además, si el enemigo no muere pero llega al extremo inferior de la
pantalla, también ocurre lo mismo: se genera un nuevo enemigo en el límite superior, con
lo que siempre habrá en pantalla 5 enemigos de forma permanente*/
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    write(0,450,460,3,"SCORE :");
    write_var(0,600,460,3,score);
    loop
        if(key(_esc)) exit(); end
        frame;
    end
end

process nave(x,y)
private
   int point;
/*xpos y ypos definen el punto desde donde saldrán los misiles,gracias a la función
get_real_point*/
   int  xpos,
   int  ypos;
end
begin
   graph=new_map(7,7,32);map_clear(0,graph,rgb(255,255,0));
   size=80;
   z=10;//Ponemos Z>0 para que los misiles se pinten por debajo
   loop
       if(key(_up))y=y-10;end
       if(key(_down))y=y+10;end
       if(key(_right))x=x+10;end
       if(key(_left))x=x-10;end
       if(y<10)y=10;end
       if(y>450)y=450;end
       if(x<20)x=20;end
       if(x>620)x=620;end
//Pulsando SPACE disparamos un tipo de misil
       if(key(_space))
/*Obtenemos del proceso actual (nave) la coordenada X e Y del punto de control 0 (es
decir, el centro de la nave)...*/
           get_real_point(0,&xpos,&ypos);
//...para posicionar el misil en esa coordenada
           misil(xpos,ypos);
/*Igualmente hacemos para situar allí donde está la nave una explosión producida por la
propulsión de nuestros misiles*/
           explosion(xpos,ypos,20);
       end
/*Pulsando CONTROL disparamos otro tipo de misil, formado por tres misiles en paralelo*/
       if(key(_control))
/*Si tiro==0 no podemos disparar. Tiro valdrá 0 justo después de haber lanzado uno de
estos misiles triples, de manera que si se continúa pulsando la tecla CTRL, no se podrá
seguir disparando. Se tendrá que dejar de pulsar la tecla CTRL, para que a la siguiente
vez que se pulse, se ejecute el else (tiro=1), y en el siguienteframe, se pueda volver a
disparar -sólo una vez, otra vez, y así-.Como se puede ver, este sistema es para
deshabilitar la posibilidad de hacer disparos infinitos con la tecla CTRL siempre
pulsada.*/
         if(tiro!=0)
           get_real_point(0,&xpos,&ypos);
           misil(xpos-10,ypos);
           misil(xpos,ypos);
           misil(xpos+10,ypos);
//Por cada misil lanzado hay una explosión de propulsión
           explosion(xpos-10,ypos,20);
           explosion(xpos,ypos,20);
           explosion(xpos+10,ypos,20);
```

```
          tiro=0;
        end
      else
/*Si se ejecuta este else es que se ha pulsado CTRL de nuevo, después de haber disparado
ya antes y haber soltado la tecla,con lo que hacemos que se pueda volver a disparar*/
          tiro=1;
      end
      frame;
   end
end

process misil(x,y)
begin
   graph=new_map(7,7,32);map_clear(0,graph,rgb(0,255,0));
   size=70;
   loop
/*Si se sale fuera de la pantalla, se acaba este proceso. */
      if(out_region(id,0))break;end
      y=y-10;
      frame;
   end
end

process explosion(x,y,size)
private
         int i;
end
begin
/*Durante 100 fotogramas (cantidad que se puede regular para hacer que la explosión dure
más o menos) se verá una explosión consistente en un cuadradito que se moverá levemente
de forma aleatoria a partir de su posición inicial, la cual vendrá dada por desde dónde
se invoque a este proceso: las explosiones pueden provenir de la eliminación de un
enemigo o bien de la propulsión de los misiles de la nave protagonista.*/
   for(i=0;i<=100;i++)
      graph=new_map(7,7,32);map_clear(0,graph,rgb(rand(100,200),rand(0,250),0));
      x=x+rand(-2,2);
      y=y+rand(-2,2);
      frame;
   end
end


process enemigo(x,y)
private
  int tiro_enemigo=0;
  int point,xpos,ypos;
end
begin
  graph=new_map(7,7,32);map_clear(0,graph,rgb(255,0,255));
  size=80;
  loop
     y=y+5;
/*Si se choca contra el misil de la nave, genero una explosión, aumento el score, genero
un nuevo enemigo en la parte superior de la pantalla relevando al presente, y éste lo
mato con el break.*/
     if(collision(type misil))
          get_real_point(0,&xpos,&ypos);
          explosion(xpos,ypos,100);
          score=score+10;
          enemigo(rand(5,635),-50);
          break;
     end
/*De igual manera, si el enemigo alcanza el extremo inferior de la pantalla, genero un
nuevo enemigo en la parte superior de la pantallarelevando al presente, el cual lo mato
con el break;*/
     if(y>680)
          enemigo(rand(5,635),-50);
          break;
```

```
      end

/*Sistema temporizador para los misiles de los enemigos. Éstos se crearán automáticamente
cada vez que la variable tiro_enemigo llegue a valer 50. Esta variable aumenta en uno su
valor a cada fotograma, y cuando llega a 50, después de haberse generado el misil
enemigo, se vuelve  a resetear a 0 para volver a empezar la cuenta. Si se quiere que los
enemigosdisparen más frecuentemente, no hay más que rebajar el límite de 50 en el if por
otro número más pequeño: ese número indicara cada cuántos fotogramas los
enemigos dispararán*/
      tiro_enemigo++;
      if(tiro_enemigo>50)
           get_real_point(0,&xpos,&ypos);
           misil_enemigo(xpos,ypos);
           tiro_enemigo=0;
      end
      frame;
  end
end

/*Exactamente igual que el proceso misil, pero con otro color y el sentido vertical del
movimiento al contrario. (De hecho, como sólo son dos pequeñas diferencias, se podría
haber escrito un solo proceso que  sirviera para generar misiles propios y enemigos, con
el uso de parámetrosadecuados).*/
process misil_enemigo(x,y)
private
  int xpos;
  int ypos;
end
begin
  graph=new_map(7,7,32);map_clear(0,graph,rgb(255,0,255));
  size=70;
  loop
       if(out_region(id,0))break;end
       y=y+10;
       frame;
  end
end
```

Si te fijas en el código anterior, verás que hemos utilizado la función **get_real_point()** para obtener el punto de control 0 de los procesos pertinentes. Ya sabemos que el punto de control 0 corresponde al centro del gráfico del proceso, con lo que haciendo servir las variables locales ***X*** e ***Y*** ya no hace falta utilizar esta función. No obstante, se ha mantenido para mostrar otra posibilidad de trabajo, y para dejar abierta una posible mejora que consistiría en crear más puntos de control en los gráficos y utilizarlo mediante **get_real_point()** (por ejemplo, crear un punto en la parte delantera de la nave y los enemigos, desde donde podrían salir los disparos).

Si juegas, verás que el protagonista no muere nunca. En otras palabras: los disparos de los enemigos (ni éstos propiamente) afectan en lo más mínimo a la salud de nuestranave. Se deja como ejercicio solucionar esto.

## *Código para crear explosiones un poco más espectaculares

A continuación veremos un código bastante ingenioso para lograr, a partir de un único gráfico (en el código de ejemplo el gráfico lo hemos generado dinámicamente, pero para que el efecto sea mayor es preferible que tegna forma de llamarada amarillo-naranja), un efecto de explosión completa, que , si bien no es nada del otro mundo, puede causar cierto asombro. El proceso "explosion()", si te gusta, lo puedes utilizar en tus propios juegos para mostrar el efecto visto.

```
import "mod_map";
import "mod_proc";
import "mod_key";
import "mod_video";
import "mod_rand";
```

```
process main()
Begin
              set_fps(24,0);
             Loop
                       If(key(_esc)) exit();End
                       explosion(160,120,size);
                       Frame;
             End
End

Process explosion(x,y,size)
Begin
      graph=new_map(40,40,32);map_clear(0,graph,rgb(255,255,0));
      Repeat
                If (size>50 AND rand(0,40)<size/8) /* Crea otras explosiones*/
                           x=x+rand(-25,45);
                           y=y+rand(-30,15);
                End
                size=size-4;  // Cambian de tamaño reduciendose
                Frame;
      Until (size<5)  // Continua hasta que sean demasiado pequeñas
End
```

***Código para efecto de caída de objetos**

```
import "mod_video";
import "mod_map";
import "mod_rand";
import "mod_proc";
import "mod_key";
import "mod_screen";
import "mod_draw";
import "mod_timers";


PROCESS Main()
private
    int idmap,idmap2;
end
BEGIN
    set_mode (320, 200, 32);
    idmap=new_map(30,30,32);map_clear(0,idmap,rgb(255,255,0));

    delete_text(0);Write(0,0,0,0,"100 sprites cayendo") ;
    FROM x = 0 TO 400; FallingObject(idmap, 100, 0, 0); END
    WHILE (!key(_ESC)) FRAME; END
    signal (TYPE FallingObject, S_KILL); WHILE (key(_ESC)) FRAME; END

    delete_text(0);Write(0,0,0,0,"100 sprites rotando") ;
    FROM x = 0 TO 100; FallingObject(idmap, 100, 0, rand(500,1000)); END
    WHILE (!key(_ESC)) FRAME; END
    signal (TYPE FallingObject, S_KILL); WHILE (key(_ESC)) FRAME; END

    delete_text(0);Write(0,0,0,0,"100 sprites transparentes") ;
    FROM x = 0 TO 100; FallingObject(idmap, 100, 4, 0); END
    WHILE (!key(_ESC)) FRAME; END
    signal (TYPE FallingObject, S_KILL); WHILE  key(_ESC): FRAME; END

    delete_text(0);Write(0,0,0,0,"100 sprites escalados") ;
    FROM x = 0 TO 100; FallingObject(idmap, 120, 0, 0); END
    WHILE (!key(_ESC)) FRAME; END
    let_me_alone();
END

PROCESS FallingObject(graph, size, flags, angleinc)
PRIVATE xspeed, yspeed, inispeed ;
```

```
BEGIN
    z      = rand (-5, 125) ;
    x      = rand (15, 305) ;
    y      = - rand (20, 100) ;
    xspeed = rand (-10, 10) ;
    yspeed = rand (-4, 0) ;
    inispeed = rand (10, 15) ;
    angle  = rand (0, 50 * angleinc) ;
    LOOP
        alpha = timer[0];

        x = x + xspeed ;
        IF (x > 305 || x < 15) xspeed = -xspeed; END
        yspeed++;
        y = y+yspeed;
        IF (yspeed < -14) yspeed = -14; END
        IF (y > 180)
            yspeed = -inispeed ;
            IF (inispeed > 1)
                inispeed--;
            ELSE
                inispeed = 15;
            END
        END

        IF (out_region(ID, 0)) y = -rand(20, 100) ; inispeed = 15 ; END

        angle = angle +angleinc ;
        FRAME ;
    END
END
```

***Código que simula la caída de nieve**

```
import "mod_video";
import "mod_rand";
import "mod_draw";
import "mod_key";
import "mod_map";
import "mod_proc";
import "mod_timers";
import "mod_math";
import "mod_grproc";

process main()
begin
  set_mode(640,480,32);
  set_fps(20,0);
  drawing_map(0,0);drawing_color(rgb(255,255,0));draw_fcircle(100,100,50);
  tierra();
  while(!key(_esc))
            nieve(rand(0,640),-5,rand(50,100),rand(1,25),rand(0,359000));
            nieve(rand(0,640),-5,rand(50,100),rand(1,25),rand(0,359000));
            nieve(rand(0,640),-5,rand(50,100),rand(1,25),rand(0,359000));
            frame;
  end
  exit();
end


process tierra()
begin
  drawing_map(0,0);
  drawing_color(rgb(255,255,255));
  draw_rect(0,450,640,480);
```

```
  graph=new_map(640,30,32);
  x=320;
  y=465;
  map_clear(0,graph,rgb(255,255,255));
  loop
          frame;
  end
end


process nieve(int xoriginal,y, size, int ampl, int fase)
begin
  graph=new_map(2,2,32);map_clear(0,graph,rgb(255,255,255));
//Mientras va bajando el copo
  repeat
            x=xoriginal+ ampl*cos(timer[0]*1000%360000 + fase);
            y=y+rand(1,10);
            frame;
  until(collision(type tierra))

//Una vez ya en el suelo
  y=y-rand(1,5);
  loop
          frame;
  end
end
```

## *Códigos que ondulan una imagen

Este código es ciertamente complicado de entender ya que en él se mezclan punteros, gestión dinámica de memoria, copia de bloques de imágenes, etc. Pero como recurso a tener en cuenta, es interesante ya que con sólo incorporar a nuestro proyecto el proceso "waves()" ya dispondremos de su funcionalidad.

```
import "mod_draw";
import "mod_key";
import "mod_video";
import "mod_rand";
import "mod_map";
import "mod_mem";
import "mod_math";
import "mod_screen";

const
       x_res=320;
       y_res=240;
end
local
       bool killed=false;
end

process main()
  private
       int box;
       int fondo, fondo2, fondo3;
  end
  begin
       set_mode(x_res,y_res,32);
       set_fps(60,0);

//Imagen de fondo cuyo extremo superior izquierdo se ondulará
       fondo=new_map(x_res,y_res,32);map_clear(0,fondo,rgb(255,255,0));
       fondo2=new_map(200,200,32);map_clear(0,fondo2,rgb(0,255,0));
       fondo3=new_map(100,100,32);map_clear(0,fondo3,rgb(0,0,255));
       map_put(0,fondo2,fondo3,200,100);
```

```
       map_put(0,fondo,fondo2,0,0);
       put_screen(0,fondo);

//Se pueden cambiar el valor de diferentes parámetros para observar qué efectos produce
       waves(50,50,8,255,100,100,15,3000,3,x_res,y_res);

/*Este cuadrado delimitará visualmente la zona que sufre ondulaciones, y se puede mover
con los cursores*/
       box=draw_rect(0,0,100,100);
           while(!key(_esc))
                       if(key(_left) and son.x>50)son.x=son.x-5;end;
                       if(key(_right) and son.x<x_res-50)son.x=son.x+5;end;
                       if(key(_up) and son.y>50)son.y=son.y-5;end;
                       if(key(_down) and son.y<y_res-50)son.y=son.y+5;end;
                       move_draw(box,son.x-50,son.y-50);
                       frame;
            end
       son.killed=true;
  end


process waves(x,y,z,alpha,int anchura,int altura,int amp,int freq,int spd,int x_res,int
y_res)
 private
      int pointer gfx;
      int ani=0,count,source;
 end
 begin
      graph=new_map(anchura,altura,32);
      gfx=alloc(sizeof(gfx)*altura);
      source=new_map(x_res+2*amp,y_res,32);
      map_block_copy(0,source,amp,0,0,0,0,x_res,y_res,0);
      for(count=0; count<altura; count++)
             gfx[count]=new_map(anchura+2*amp,1,32);
      end
      while(!killed)
           ani=ani+spd;
           map_clear(0,graph,0);
           for(count=0;count<altura;count++)
                    map_clear(0,gfx[count],0);
  map_block_copy(0,gfx[count],0,0,source,x-anchura/2,y-altura/2+count,anchura+2*amp,1,0);
         map_put(0,graph,gfx[count],(anchura/2)+get_distx((ani+y+count)*freq,amp),count);
           end
           frame;
      end
      unload_map(0,source);
      for(count=0; count<altura; count++)  unload_map(0,gfx[count]);  end
 end
```

A continuación presento otro código que persigue un objetivo idéntico: ondular una imagen, pero utilizando un método diferente: haciendo uso de la función **map_block_copy()** en un bucle, y nada más. Con muy poco esfuerzo, podrías modificar el código para poderlo reconvertir en un proceso autónomo (al estilo del proceso "waves()" del ejemplo anterior) para aplicar la ondulación a cualquier gráfico de cualquier juego de forma inmediata. Para probarlo, necesitarás una imagen llamada "a.png".

```
Import "mod_video";
Import "mod_map";
Import "mod_key";
Import "mod_screen";
Import "mod_math";

process main()
Private
   int angulo,altura,desfase;
   int radio=20, omega=5000, delta=5000;
   int grafico, grafico2;
end
```

```
Begin
   set_mode(640,480,32);
   grafico=load_png("a.png");
   grafico2=new_map(radio*2+map_info(0,grafico,g_width),map_info(0,grafico,g_height),32);
   loop
       angulo=desfase;
       from altura=0 to map_info(0,grafico,g_height);
map_block_copy(0,grafico2,radio+get_distx(angulo,radio),altura,grafico,0,altura,map_info(
0,grafico,g_width),1,128);
                  angulo=angulo+omega;
       end
       desfase=desfase+delta;
       if(angulo>360000) angulo=0; end
       if(desfase>360000) desfase=0; end
       if(key(_esc)) break; end
       put_screen(0,grafico2);
       frame;
   end
end
```

## *Código que provoca temblores en una imagen

Un código curioso es el siguiente (necesitarás una imagen llamada “a.png” para probarlo), que consigue dotar de un efecto “temblor” a la imagen de un proceso. La idea del código anterior es sencilla. Se muestra un proceso, y cuando se apreta la tecla “e”, aparece el efecto temblor. Para que desaparezca éste, se ha de apretar la tecla “f”. El proceso “efecto_temblor()” es el más interesante: recoge los valores de la ***X*** e ***Y*** del proceso “miproceso()” y también recibe como parámetro un número que será la magnitud del temblor.A partir de aquí, lo único que se hace para simular ese temblor es variar la ***X*** e ***Y*** aleatoriamente de este nuevo proceso, a partir de la ***X*** e ***Y*** originales y dependiendo de la magnitud fijada. Pero todavía no se ve nada en pantalla. Lo interesante del asunto está en que a cada movimiento del gráfico, se hace una captura de pantalla y ésta es la que se asignará al gráfico del proceso “efecto_temblor()”, por lo que en realidad en el programa estaremos viendo dos gráficos cada vez, el del “miproceso()”, inamovible, y el de “efecto_temblor()”, el cual en cada iteración cambiará. Importante recalcar que después de haber visionado por pantalla el gráfico correspondiente a una iteración del temblor, se descarga de memoria para proseguir con la siguiente captura, ya que si no el programa a los pocos segundos se ralentizaría hasta el extremo.

```
Import "mod_video";
Import "mod_map";
Import "mod_screen";
Import "mod_key";
Import "mod_rand";
Import "mod_proc";

process main()
begin
set_mode(640,480);
miproceso();
while(!key(_esc))
          frame;
end
let_me_alone();
end

process miproceso()
begin
graph=load_png("a.png");
x=320;
y=240;
loop
  if(key(_e)) efecto_temblor(15,x,y);end
  frame;
end
end
```

```
Process efecto_temblor(int magnitud,a,b)
private
          int mapatemblor;
end
begin
z=-1;
while(!key(_f))
   x=a+rand(-magnitud,magnitud);
   y=b+rand(-magnitud,magnitud);
   mapatemblor=get_screen();
   graph=mapatemblor;
   frame;
   unload_map(0,mapatemblor);
end
end
```

## *Realización de diferentes transiciones entre imágenes

El siguiente código de ejemplo, el cual utiliza procesos reutilizables, consiste en pasar de una imagen a otra mediante una cortina horizontal. Para probarlo has de utilizar dos imágenes PNG: "a.png" y "b.png". Una vez probado el ejemplo, si te gusta, para poderlo utilizar en tu juego o programa, deberás copiar a tu código los 2 procesos : "transicion()" y "pixel()". Y cuando quieras utilizar el efecto, tendrás que escribir: *transicion(int grafico1,int grafico2,int velocidad);* ¡No te olvides de hacer pruebas!

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";
Import "mod_timers";

process main()
Private
          int ancho_mapa;
          int alto_mapa;
end
Begin
          set_mode(640,480,32);
          set_fps(60,0);
          //iniciamos la transición
          transicion(load_png("sol.png"),load_png("fondo.png"),10);
/* descomentar lo de abajo si en vez de otro gráfico queremos que se vuelva negra la
pantalla*/
//        transicion(load_png("a.png"),0,10);
End


/*Estos dos procesos son los que tienes que copiar a tu juego o programa si quieres usar
este efecto */
Process transicion(fixero1,fixero2,velocidad)
Private
          int ancho_mapa;
          int alto_mapa;
          int tipo=0;
end
Begin
          ancho_mapa=map_info(0,fixero1,g_width);
          alto_mapa=map_info(0,fixero1,g_height);
          While(y<alto_mapa/2+1)
                    If(tipo==1) tipo=0; Else tipo=1; End
                    pixel(fixero1,y,ancho_mapa,velocidad,tipo);
                    pixel(fixero1,alto_mapa-y,ancho_mapa,velocidad,tipo);
                    y++;
                    Frame(velocidad*3);
```

```
		End
		timer[0]=0;
		While(timer[0]<300)
			Frame;
		End
		y=0; x=0;
		If(fixero2==0) fixero2=new_map(640,480,8); End
		While(y<alto_mapa/2+1)
			If(tipo==1) tipo=0; Else tipo=1; End
			pixel(fixero2,y,ancho_mapa,velocidad,tipo);
			pixel(fixero2,alto_mapa-y,ancho_mapa,velocidad,tipo);
			y++;
			Frame(velocidad*3);
		End
		While(NOT(key(_esc)))
		Frame;
		End
End

/*este proceso que no se te olvide, van juntos*/
Process pixel(fixero,y,ancho_mapa,velocidad,tipo);
Private
		color;
Begin
		graph=new_map(1,1,8);
		drawing_map(0,graph); //Las funciones draw_* las veremos en el siguiente apartado
		drawing_color(254);
		draw_fcircle(0,0,1);
		If(tipo==0)
			While(x<ancho_mapa-1)
				x++;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x++;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x++;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x++;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x++;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				Frame(velocidad);
			End
		End
		If(tipo==1)
			x=ancho_mapa;
			While(x>1)
				x--;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x--;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x--;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x--;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				x--;
				color=map_get_pixel(0,fixero,x,y);
				map_put_pixel(0,father.graph,x,y,color);
				Frame(velocidad);
			End
```

```
          End
End
```

Otra transición que se puede conseguir es la “cascada”. Un ejemplo:

```
Import "mod_video";
Import "mod_map";
Import "mod_draw";
Import "mod_key";
Import "mod_screen";
Import "mod_rand";

Const
   tam=5; //cambiar para modificar el tamaño de los "fideos", cuanto mas grandes mejor rendimiento
End
Global
  Byte empieza;
  int fondo1;
  int fondo2;
end

process main()
Private
   int xf=tam;
end
Begin
   set_mode(640,480,32);
   fondo1=load_png("sol.png");
   fondo2=load_png("fondo.png");
   put_screen(0,fondo1);
   While(xf>-690)
      xf=xf-tam;
      fideo(xf);
      Frame;
   End
   empieza=1;
 End

Process fideo(cual)
  Private int vy; end
  Begin
   flags=128;
   y=240;x=-cual+(tam/2);
   vy=rand(0,10);
   graph=new_map(tam,480,32);
   map_put(0,graph,fondo2,cual,0);
  Repeat
     Frame;
  Until(empieza)
  Repeat
     y=y+vy;
     vy++;
     Frame;
  Until(out_region(id,0))//Hasta que el gráfico salga fuera de la pantalla.Consultar out_region()
  unload_map(0,graph);
End
```

Finalmente, aquí transcribo un código fuente que muestra el funcionamiento de cinco efectos: espiral, persiana horizontal, persiana vertical, cortina horizontal y cortina vertical. La idea del código es que los procesos que corresponden a cada uno de estos efectos sean inmediatamente transportables a otros códigos fuente que podamos hacer, de manera que tengamos esta funcionalidad ya lista para funcionar. Si te fijas, para conseguir estos efectos las únicas funciones que hemos necesitado han sido **map_get_pixel()** y **put_pixel()** (y **map_info()**).Para poderlos visualizar necesitarás tener un FPG llamado “graficos.fpg” con dos gráficos en su interior, de códigos 001 y 002 (es aconsejable que el gráfico 002 sea de dimensiones bastante menores que 001, para una mayor comodidad en la visualización del efecto):

```
/*      sp_put_ESPIRAL        ( fpg, grafico, x, y, modo );
```

```
        sp_put_PERSIANA_HORIZ ( fpg, grafico, x, y, modo );
        sp_put_PERSIANA_VERT  ( fpg, grafico, x, y, modo );
        sp_put_CORTINA_HORIZ  ( fpg, grafico, x, y, modo, velocidad );
        sp_put_CORTINA_VERT   ( fpg, grafico, x, y, modo, velocidad );        */
Import "mod_video";
Import "mod_map";
Import "mod_timers";
Import "mod_screen";

Process Main()
begin
    set_fps(25, 0);
    set_mode(800,600,32);
    load_fpg("graficos.fpg");
    loop
 //El while posibilita la visualización de los efectos
        sp_put_ESPIRAL         (0, 1, 50, 20, 0);     timer=0; while(timer<200) frame; end
        sp_put_ESPIRAL         (0, 2, 60, 40, 1);     timer=0; while(timer<200) frame; end
        sp_put_PERSIANA_HORIZ  (0, 1, 50, 20, 0);     timer=0; while(timer<200) frame; end
        sp_put_PERSIANA_HORIZ  (0, 2, 60, 40, 1);     timer=0; while(timer<200) frame; end
        sp_put_PERSIANA_VERT   (0, 1, 50, 20, 0);     timer=0; while(timer<200) frame; end
        sp_put_PERSIANA_VERT   (0, 2, 60, 40, 1);     timer=0; while(timer<200) frame; end
        sp_put_CORTINA_HORIZ   (0, 1, 50, 20, 0, 10); timer=0; while(timer<200) frame; end
        sp_put_CORTINA_HORIZ   (0, 2, 60, 40, 1, 10); timer=0; while(timer<200) frame; end
        sp_put_CORTINA_VERT    (0, 1, 50, 20, 0, 10); timer=0; while(timer<200) frame; end
        sp_put_CORTINA_VERT    (0, 2, 60, 40, 1, 10); timer=0; while(timer<200) frame; end
    end
end

//-- Efecto 1: - Fundido en espiral ----------==oOo==---------- by !Deemo -<

process sp_put_ESPIRAL(fpg, grafico, x, y, modo)
private
    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
     int Espiral[49]=  0,0, 1,0, 2,0, 3,0, 4,0, 4,1, 4,2, 4,3,4,4, 3,4, 2,4, 1,4, 0,4, 0,3, 0,2,
0,1,1,1, 2,1, 3,1, 3,2, 3,3, 2,3, 1,3, 1,2, 2,2;
end
begin
    alto=  map_info(fpg, grafico, g_height);
    ancho= map_info(fpg, grafico, g_width);

    if(modo==0)
                        ini=0; fin=50; inc=2;
    else
                        ini=48; fin=-2; inc=-2;
    end

    for(c=ini; c<>fin; c=c+inc)
        for(gy=0; gy<alto; gy=gy+5)
            for(gx=0; gx<ancho; gx=gx+5)
                pixel= map_get_pixel(fpg, grafico, gx+Espiral[c], gy+Espiral[c+1]);
                if(pixel<>0) put_pixel(x+gx+Espiral[c], y+gy+Espiral[c+1], pixel); end
            end
        end
        frame;
    end
end


//-- Efecto 2: - Persiana horizontal ----------==oOo==---------- by !Deemo -<

process sp_put_PERSIANA_HORIZ(fpg, grafico, x, y, modo)
private
    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
end
begin
    alto=  map_info(fpg, grafico, g_height);
    ancho= map_info(fpg, grafico, g_width);

    if(modo==0)
                        ini=0; fin=5; inc=1;
    else
                        ini=5; fin=0; inc=-1;
    end

    for(c=ini; c<>fin; c=c+inc)
        for(gy=0; gy<alto; gy++)
```

```
            for(gx=0; gx<ancho; gx=gx+5)
                pixel= map_get_pixel(fpg, grafico, gx+c, gy);
                if(pixel<>0) put_pixel(x+gx+c, y+gy, pixel); end
            end
        end
        frame;
    end
end


//-- Efecto 3: - Persiana vertical ------------==oOo==------------ by !Deemo -<

process sp_put_PERSIANA_VERT(fpg, grafico, x, y, modo)
private
    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
end
begin
    alto=  map_info(fpg, grafico, g_height);
    ancho= map_info(fpg, grafico, g_width);

    if(modo==0)
                                    ini=0; fin=5; inc=1;
    else
                                    ini=5; fin=0; inc=-1;
    end

    for(c=ini; c<>fin; c=c+inc)
        for(gy=0; gy<alto; gy=gy+5)
            for(gx=0; gx<ancho; gx++)
                pixel= map_get_pixel(fpg, grafico, gx, gy+c);
                if(pixel<>0) put_pixel(x+gx, y+gy+c, pixel); end
            end
        end
        frame;
    end
end

//-- Efecto 4: - Cortina horizontal ------------==oOo==----------- by !Deemo -<

process sp_put_CORTINA_HORIZ(fpg, grafico, x, y, modo, velocidad)
private
    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
end
begin
    alto=  map_info(fpg, grafico, g_height);
    ancho= map_info(fpg, grafico, g_width);

    if(modo==0)
                        ini=0; fin=ancho+1; inc=1;
    else
                        ini=ancho-1; fin=0; inc=-1;
    end

    for(gx=ini; gx<>fin; gx=gx+inc)
        for(gy=0; gy<alto; gy=gy+2)
            pixel= map_get_pixel(fpg, grafico, gx, gy);
            if(pixel<>0) put_pixel(x+gx, y+gy, pixel); end
            pixel= map_get_pixel(fpg, grafico, ancho-gx, gy+1);
            if(pixel<>0) put_pixel(x+ancho-gx, y+gy+1, pixel); end
        end
        frame(velocidad);
    end
end

//-- Efecto 5: - Cortina vertical ------------==oOo==------------ by !Deemo -<

process sp_put_CORTINA_VERT(fpg, grafico, x, y, modo, velocidad)

private
    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
end
begin
    alto=  map_info(fpg, grafico, g_height);
    ancho= map_info(fpg, grafico, g_width);

    if(modo==0)
                        ini=0; fin=alto+1; inc=1;
```

```
    else
                        ini=alto-1; fin=0; inc=-1;
    end

    for(gy=ini; gy<>fin; gy=gy+inc)
        for(gx=0; gx<ancho; gx=gx+2)
            pixel= map_get_pixel(fpg, grafico, gx, gy);
            if(pixel<>0) put_pixel(x+gx, y+gy, pixel); end
            pixel= map_get_pixel(fpg, grafico, gx+1, alto-gy);
            if(pixel<>0) put_pixel(x+gx+1, y+alto-gy, pixel); end
        end
        frame(velocidad);
    end
end
```

***Simulación de la escritura de una antigua máquina de escribir**

```
import "mod_key";
import "mod_video";
import "mod_proc";
import "mod_text";
import "mod_string";

const
   NUMLINEAS=5;
end

process main()
private
    string texto= "Debería estar prohibido
                   haber vivido y no haber amado.
                   Por eso, tírame un beso,
                   que he sido preso de nuestro encierro.
                   Jugar con fuego.";
end
begin
    set_mode(640,480,32);
    escribe_texto(texto);
    repeat
        frame;
    until(key(_esc))
end

process escribe_texto(string strin)
private
    string lineas[NUMLINEAS-1];
    int lineaactual=0;
    string caracter;
    int i;
end
begin
/*Este bucle sirve en general para rellenar en cada elemento del vector  lineas[] una
cadena de caracteres que corresponde a cada una de las líneas del texto. Cada una de
estas lineas se tratará posteriormente en el programa como si también fuera una especie
de vector de caracteres.*/
   i=0;
   while (i<=len(strin))
          caracter=strin[i];
/*En Windows el salto de linea viene dado por dos caracteres, chr(10) y chr(13). Si se
detecta el primero, lo que hago es recoger el segundosin añadir nada al vector lineas[],
cambiar de línea y entonces seguir con
la siguiente iteración normal.*/
          if(caracter==chr(10))
                    i++;caracter=strin[i];
                    lineaactual++;
```

```
          else
/*Aquí es donde realmente añado el siguiente carácter a la línea que corresponda,como si
cada elemento del vector lineas (es decir, lineas[0], lineas[1]...) fuera a su vez un
vector de caracteres.*/
                       lineas[lineaactual]=lineas[lineaactual]+caracter;
          end
           i++;
   End

//Para cada línea
   for(lineaactual=0; lineaactual<NUMLINEAS; lineaactual++)
       //Para cada carácter de esa línea
       For(i=0; i<=len(lineas[lineaactual]); i++)
           //El switch determina la posición "y" del texto
          switch(lineaactual)
/*Imprimo un carácter cada vez.Trato cada caracter como si fuera un elemento "char"
dentro de los vectores de elementos "char" que representa que son lineas[0], lineas[1],
lineas[2]...aunque en realidad en vez de vectores de carácteres son cadenas; pero los
conceptos son fácilmente intercambiables*/
                     case 0: write(0,10+14*i,340,0,lineas[lineaactual][i]); end
                     case 1: write(0,10+14*i,370,0,lineas[lineaactual][i]); end
                     case 2: write(0,10+14*i,400,0,lineas[lineaactual][i]); end
                     case 3: write(0,10+14*i,430,0,lineas[lineaactual][i]); end
                     case 4: write(0,10+14*i,460,0,lineas[lineaactual][i]); end
          end
          frame;
       End
   End
   //No pasa nada hasta que apreto la tecla CTRL...
   while(!key(_control)) frame; end
   //...momento en el que borro todo el texto
   delete_text(all_text);
end
```

## *El juego del "disparaletras"

No se trata ahora de llenar este documento de códigos curiosos y/o interesantes, porque sería un libro inabarcable. Los límites los impone sólo la imaginación de cada uno. Pero sí que me gustaría presentar este pequeño juego como ejemplo de que con una idea realmente simple (y un código no muy difícil)se puede lograr un entretenimiento ameno. La intención es mostrar que no tenemos porqué ceñirnos siempre a los moldes tradicionales (carreras de coches, juegos de lucha, juegos tradicionales de mesa, plataformas,etc) sino que un juego original puede surgir a partir de cualquier buena idea.

El juego consiste en un menú inicial, donde se configuran los parámetros del juego, para seguidamente “disparar letras”. Es decir, en la pantalla saldrán aleatoriamente letras -y en posiciones aleatorias también- y el jugador tendrá que pulsar la tecla correspondiente dentro de un tiempo de reacción fijado. Si logra ser lo suficientemente rápido y acertar la letra, ésta explotará. Al final del juego se estable el recuento total de letras acertadas. Así de simple.

```
import "mod_draw";
import "mod_key";
import "mod_video";
import "mod_rand";
import "mod_proc";
import "mod_text";
import "mod_timers";
import "mod_map";

GLOBAL
  int intentos=1;
  int aciertos=0;
  int tiempo=5;
  int oportunidades=0;
end
```

```
process main()
BEGIN
          set_mode(320,240,32,mode_2xscale);
          set_fps(60,1);
          menu();
END

process menu()
private
     int opcion=1;
     int id_texto[3];
     int gatillo;
end
begin
     let_me_alone();
     delete_text(0);
     write(0,160,10,4,"Jugador, bienvenido a cazaletras");
     write(0,160,30,4,"Instrucciones: durante el juego pulsa la tecla");
     write(0,160,40,4,"de la letra que aparece en la pantalla.");
     write(0,160,50,4,"Selecciona el número de intentos o el tiempo");
     write(0,160,60,4,"con las teclas derecha e izquierda y pulsa");
     write(0,160,70,4,"enter para empezar. Buena suerte.");
     write(0,150,90,4,"intentos:");
     write_var(0,185,90,4,intentos);
     write(0,150,100,4,"tiempo:");
     write_var(0,185,100,4,tiempo);
     write(0,150,120,4,"aciertos:");
     write_var(0,185,120,4,aciertos);
     loop
/*Sistema muy ingenioso para generar un menú con la opción seleccionada realzada en otro
color. Simplemente se trata de, antes de escribir  el  texto de la opción elegida,
cambiarle el color con la función set_text_color y escribir seguidamente dicha opción de
nuevo en pantalla con el nuevo color. La variable gatillo sirve para controlar el pulsado
de las teclas y evitar que una sola pulsación demasiado larga el programa la entienda
como muchas pulsaciones: mientras gatillo=0 se reconocerán las pulsaciones de teclas y
cuando valga 1 no (de hecho, siempre, justo después de pulsar una tecla gatillo pasa a
valer 1 por eso). Solamente volverá a valer 0 en el momento que la variable del sistema
scan_code valga 0, es decir, que no se detecte ninguna pulsación actual de ninguna tecla,
y por tanto, se pueda volver a aceptar otra vez nuevas pulsaciones.*/
          if(opcion==1)
                    set_text_color(rgb(255,0,0));
          else
                    set_text_color(rgb(0,255,0));
          end
          id_texto[0]=write(0,160,150,4,"modo reflejos");
          if(opcion==2)
                    set_text_color(rgb(255,0,0));
          else
                    set_text_color(rgb(0,255,0));
          end
          id_texto[1]=write(0,160,160,4,"modo time atack");
          if(opcion==3)
                    set_text_color(rgb(255,0,0));
          else
                    set_text_color(rgb(0,255,0));
          end
           id_texto[2]=write(0,160,170,4,"salir");
           frame;
           delete_text(id_texto[0]);
           delete_text(id_texto[1]);
           delete_text(id_texto[2]);

/*Según en la opcion del menú donde estemos, y el cursor (derecho o izquierdo) que
pulsemos, así afectará a la configuración del juego (número de intentos y tiempo).
Fijarse que el nº de intentos sólo puede ir entre 1 y 10, y el tiempo entre 5 y 30.*/
  IF(key(_right) and intentos<10 and opcion==1 and gatillo==0) intentos++; gatillo=1; END
  IF(key(_left) and intentos>1 and opcion==1 and gatillo==0) intentos--; gatillo=1; END
```

```
  IF(key(_right) and tiempo<30 and opcion==2 and gatillo==0)tiempo=tiempo+5;gatillo=1;END
  IF(key(_left) and tiempo>5 and opcion==2 and gatillo==0) tiempo=tiempo-5;gatillo=1; END
           if(key(_down) and gatillo==0 and opcion<3) opcion++; gatillo=1; end
           if(key(_up) and gatillo==0 and opcion>1) opcion--; gatillo=1; end
           if(gatillo==1 and scan_code==0) gatillo=0; end
/*Entramos para jugar*/
           if(key(_enter))
                       if(opcion==1) cazar(1); end
                 if(opcion==2) cazar(2); end
                 if(opcion==3) exit(0,""); end
           end
     end  //loop
end


PROCESS cazar(modo)
private
     int time;
     int id_letra;
end
BEGIN
  let_me_alone();
  delete_text(0);
  set_text_color(rgb(255,255,255));
  aciertos=0;
  oportunidades=0;
/*Las líneas que siguen sirven para causar un efecto de cuenta atrás antes de empezar el
juego: salen una detrás de otra las palabras “preparado”, “3”,”2” y “1”.*/
  timer[1]=0;
  time=0;
  write(0,160,120,4,"preparado");
  while(timer[1]<100) frame; end
  timer[1]=0;
  delete_text(0);
  write(0,160,120,4,"3");
  while(timer[1]<100) frame; end
  timer[1]=0;
  delete_text(0);
  write(0,160,120,4,"2");
  while(timer[1]<100) frame; end
  timer[1]=0;
  delete_text(0);
  write(0,160,120,4,"1");
  while(timer[1]<100) frame; end
  timer[1]=0;
  delete_text(0);
/*Aquí comienza el juego de verdad*/
  if(modo==1)
           LOOP
                     time=-25*oportunidades*oportunidades+525*oportunidades+100;
                     IF(timer[1]>=time)
                               signal(id_letra,s_kill);
                               frame;
                               delete_text(0);
                              id_letra=letras(rand(1,26));
                              oportunidades++;
                     END
                     IF(oportunidades==intentos+1)
                               menu();
                              delete_text(0);
                              break;
                     END
                     frame;
           end //loop
  end //if
  if(modo==2)
                id_letra=letras(rand(1,26));
                write_var(0,160,5,4,time);
```

```
                loop
                            time=tiempo-timer[1]/100;
                            if( not exists(id_letra))
                                  id_letra=letras(rand(1,26));
                            end
                           if(timer[1]>tiempo*100) menu(); end
                            frame;
                end
  end // if
end


/*El código de este proceso es evidente*/
PROCESS letras(letra)
PRIVATE
   int pulsar;
   int id_letra;
end
BEGIN
   x=rand(10,310);
   y=rand(10,230);
   SWITCH (letra)
      case 1: id_letra=write(0,x,y,4,"a"); pulsar=_a; END
      case 2: id_letra=write(0,x,y,4,"b"); pulsar=_b; END
      case 3: id_letra=write(0,x,y,4,"c"); pulsar=_c; END
      case 4: id_letra=write(0,x,y,4,"d"); pulsar=_d; END
      case 5: id_letra=write(0,x,y,4,"e"); pulsar=_e; END
      case 6: id_letra=write(0,x,y,4,"f"); pulsar=_f; END
      case 7: id_letra=write(0,x,y,4,"g"); pulsar=_g; END
      case 8: id_letra=write(0,x,y,4,"h"); pulsar=_h; END
      case 9: id_letra=write(0,x,y,4,"i"); pulsar=_i; END
      case 10: id_letra=write(0,x,y,4,"j"); pulsar=_j; END
      case 11: id_letra=write(0,x,y,4,"k"); pulsar=_k; END
      case 12: id_letra=write(0,x,y,4,"l"); pulsar=_l; END
      case 13: id_letra=write(0,x,y,4,"m"); pulsar=_m; END
      case 14: id_letra=write(0,x,y,4,"n"); pulsar=_n; END
      case 15: id_letra=write(0,x,y,4,"o"); pulsar=_o; END
      case 16: id_letra=write(0,x,y,4,"p"); pulsar=_p; END
      case 17: id_letra=write(0,x,y,4,"q"); pulsar=_q; END
      case 18: id_letra=write(0,x,y,4,"r"); pulsar=_r; END
      case 29: id_letra=write(0,x,y,4,"s"); pulsar=_s; END
      case 20: id_letra=write(0,x,y,4,"t"); pulsar=_t; END
      case 21: id_letra=write(0,x,y,4,"u"); pulsar=_u; END
      case 22: id_letra=write(0,x,y,4,"v"); pulsar=_v; END
      case 23: id_letra=write(0,x,y,4,"w"); pulsar=_w; END
      case 24: id_letra=write(0,x,y,4,"x"); pulsar=_x; END
      case 25: id_letra=write(0,x,y,4,"y"); pulsar=_y; END
      case 26: id_letra=write(0,x,y,4,"z"); pulsar=_z; END
  END
/*Mientras no se pulse la tecla adecuada, estaremos en un bucle infinito*/
  loop
           IF(key(pulsar))
                    aciertos++;
                    explosion(x,y);
                    break;
           end
      frame;
  end
  delete_text(id_letra);
END


/*La explosión son dos círculos concéntricos que van creciendo de tamaño hasta
desaparecer*/
PROCESS explosion(x,y)
BEGIN
  graph=new_map(11,11,8);
  drawing_map(0,graph);
```

```
  drawing_color(rand(1,255));
  draw_circle(5,5,5);
  drawing_color(rand(1,255));
  draw_circle(5,5,2);
  REPEAT
          size=size+10;
    frame;
  UNTIL(size>200)
END
```

## *El juego de disparar a la lata bailarina

A continuación presento el código de un "shooter": un juego de hacer puntería mediante el puntero del ratón:

```
Import "mod_video";
Import "mod_text";
Import "mod_key";
Import "mod_map";
Import "mod_proc";
Import "mod_draw";

Process Main()
Begin
    set_mode(640,480,32);
    can(320,240);
    p_mouse();
    while (!key(_esc)) frame; end
    let_me_alone();
end

process p_mouse()
begin
    z = -1;
    graph = new_map (4,4,32); map_clear(0,graph,RGB(255,0,0));
    while (!key(_esc))
         x= mouse.x;
         y= mouse.y;
         frame;
    end
    unload_map(0,mouse.graph);
end

process can( int disparo_x, int disparo_y)
private
    int var_x;
    int var_y;
    float const_a ;
    float const_b ;
    float const_c;
    int sigue_pulsado = 0;
    int puntos;
    float valor_dist;
    int id_mouse;
    int direction = 1;
end
begin
    size = 55;
    var_x  = 0;
    const_a = 0.1;
    const_b = -8;
    const_c = 7;
    graph = new_map(50,100,32);map_clear(0,graph,rgb(0,255,0));
    write(0,20,20,0,"puntos:");
```

```
    write_var(0,100,20,0,puntos);
    while (!key(_esc))
        angle=(angle - 10000)%360000;
        var_x++;
        var_y = const_a*pow(var_x,2) +const_b*var_x + const_c ;
        x = disparo_x + direction*var_x;
        y = (disparo_y + var_y);
        id_mouse = collision (type p_mouse);
        if (id_mouse)
            if(mouse.left)
                if (!sigue_pulsado)
                            sigue_pulsado = 1;
                            puntos=puntos+100;
                            disparo_x = x;
                            disparo_y = y;
                             var_x =0;
                            if (x < id_mouse.x)
                                direction = -1;
                             else
                                direction = 1;
                            end
                end
                valor_dist =get_dist(id_mouse);
            else
                sigue_pulsado = 0;
            end
        end
        if (y > 480)
            let_me_alone();
            break;
        end
        Frame;
    end
    unload_map(0,graph);
end
```

## *El juego de las "lonas rebota-pelotas"

He aquí otro juego muy interesante, y adictivo.

```
import "mod_key";
import "mod_mouse";
import "mod_map";
import "mod_screen";
import "mod_proc";
import "mod_grproc";
import "mod_text";
import "mod_video";
import "mod_math";
import "mod_draw";


declare Process PINTA_LINEA(INT mx1, INT my1, INT mx2, INT my2, int fam)
public
          int angulo;
          int family;
end
end

Process Main()
Begin
set_mode(640,480,32);
set_fps(60,0);
definemouse();
```

```
Bola();
While (NOT key(_esc))
          if (key(_space))
                    let_me_alone();
                    clear_screen();
                    delete_text(all_text);
                    frame(200);
                    definemouse();
                    bola();
          end
          Frame;
End
exit("",0);
End


Process definemouse()
Private
          INT pulsado=0;
          INT mx1=0; my1=0;
          INT mx2=0; my2=0;
          int famcnt = 0;
End
Begin
mouse.x=270;
mouse.y=240;
pulsado=0;
mx1 = -1; my1 = -1;

Loop
          graph = new_map(8,8,32);
          drawing_map(file,graph);
          drawing_color(rgb(255,255,255));
          draw_box(0,0,8,8);

          If (mouse.left)
                    if (!pulsado) famcnt++; end
                    pulsado=1;
                    if (mx1!=-1 && my1!=-1)
                              PINTA_LINEA(mx1,my1,mouse.x,mouse.y, famcnt);
                    end
                    mx1 = mouse.x; my1 = mouse.y;
                    frame;
          else
                    pulsado=0;
                    mx1 = -1; my1 = -1;
          End

          x=mouse.x;
          y=mouse.y;
          Frame;
End
End


Process BOLA()
Private
          PINTA_LINEA colision;
          int angulo;
          int family;
          int angle_delta;
          int angle_inc;
End
Begin
graph = new_map(32,32,32);
drawing_map(file,graph);
draw_fcircle(16, 16, 16);
x = 320; y = 240;
```

```
angulo=45000;
WRITE_VAR(0,0,10,10, angulo);
WRITE_VAR(0,0,20,10, family);
Loop
          //Choque con Paredes
          IF (x<16 or X>624)
                    angulo=(180000-angulo)%360000;
                    if (angulo<0)angulo+=360000;end
          end
          IF (y<16 or Y>464)
                    angulo=(360000-angulo)%360000;
                    if (angulo<0)angulo+=360000;end
          end
          XADVANCE(angulo, 2);
          colision = COLLISION(TYPE PINTA_LINEA);
          IF (colision)
                    angulo = choque( colision.angulo, angulo);
                    family = colision.family;
                    colision = get_id(0);
                    while (colision=get_id(type PINTA_LINEA))
                              if (colision.family == family)
                                        SIGNAL(colision, s_kill);
                              end
                    end
          END
          angulo%=360000;
          if (angulo<0)angulo+=360000;end
          FRAME;
END
END

Process PINTA_LINEA(INT mx1, INT my1, INT mx2, INT my2, int fam)
Private
          int texto;
end
Begin
family = fam;
graph=new_map(abs(mx1-mx2)+2, abs(my1-my2)+2,32);
drawing_color(rgb(255,255,255));
drawing_map(0,graph);
set_point(0,graph, 0, 0,0);

//el punto Y más bajo indica desde donde empieza el ángulo
//(así, siempre será entre 0 y 180°)
if (my1<my2)
          angulo = fget_angle(mx2,my2,mx1,my1);
else
          angulo = fget_angle(mx1,my1,mx2,my2);
end
texto=write(0,mx1, my1,0,angulo);

if (mx1<mx2)
          x = mx1;
          mx2=mx2-mx1; mx1=0;
elseif(mx1>mx2)
          x = mx2;
          mx1=mx1-mx2; mx2=0;
else
          x = mx1;
          mx1=0; mx2=0;
end

if (my1<my2)
          y = my1;
          my2=my2-my1; my1=0;
elseif(my1>my2)
          y = my2;
          my1=my1-my2; my2=0;
```

```
else
          y = my1;
          my1=0; my2=0;
end

//dibuja la línea en el gráfico creado
draw_line(mx1, my1, mx2, my2);

LOOP
          FRAME;
END

onexit:
          delete_text(texto);
END


function choque(int angLinea, int angBola)
private
          int d;
end
begin
/*Calculamos el ángulo de incidencia. La variable 'd' vendría a ser el ángulo que tiene
la bola en un sistema de coordenadas en el cual en ángulo de la línea es 0.*/
d = angBola - angLinea;
if( (d == 0) || (abs(d) == 180000) )
//Caso especial en que rebotamos con el borde de la línea. Podría ser un rebote aleatorio
          angBola += 180000;
else
//Descomponemos el vector de velocidad de la bola en sus dos componentes
          x = 1000 * cos(d);
          y = 1000 * sin(d);
/*Calculamos el ángulo correspondiente a (x, -y). Como la coordenada y de pantalla crece
de arriba hacia abajo, a fget_angle() le pasamos -(-y), es decir, y)*/
          angle = fget_angle(0, 0, x, y);
/*angle sería el ángulo de la bola en el sistema de coordenadas transformado, volvemos a
sumar el ángulo de la línea para volver al sistema de coordenadas original*/
          angBola = angLinea + angle;
end

// Normalizamos el ángulo resultante
while(angBola >= 360000) angBola -= 360000; end
while(angBola < 0) angBola += 360000; end

return angBola;
end
```

# EPILOGO (extraído de la carta abierta de Emilio de Paz, http://www.alcachofasoft.com)

A los desarrolladores independientes:

¿Por qué hacemos videojuegos?

No se trata de "Hacer el mejor motor del mercado".Ni se trata de "Estamos haciendo un juego que va a vender un montón". No se trata de "Sorprender a los demás con la idea que a nadie se le ha ocurrido".Aquí se trata de "¿Qué hay que usar con la polea para subir el cubo?".Aquí hablamos de "¡Mierda! ¡Me han matado! Voy a intentarlo otra vez, ahora no me va a pasar".

En muchas ocasiones, nosotros mismos, los creadores de videojuegos, no somos conscientes de lo que estamos haciendo. El proceso de desarrollo nos abarca y, estamos tan inmersos en él, que quizá no nos damos cuenta del aspecto global.¿Por qué hacemos videojuegos? ¿De dónde sale ese impulso de CREAR?

Miremos hacia atrás, a nuestra infancia, adolescencia o juventud, y pensemos en los primeros juegos que cayeron en nuestras manos. Los cargamos en nuestros viejos ordenadores y nos pusimos a jugar con ellos... ¿Qué sentíamos? No era "Qué bien optimizada está la memoria RAM". No era "¿Cómo habrán conseguido el naranja en un Spectrum?". Era el paso a un mundo mágico. Era Alicia a través del espejo. Es el mundo de las sensaciones, de las emociones. De vernos atrapados por un juego. Sin más. Sin "El personaje tiene más polígonos que el de Tomb Raider". Sin "Las sombras molan un mazo porque se arrojan a tiempo real". Sin "Vamos a distribuirlo en Francia, Alemania y Djibuti".

¿Qué es la "adicción", la "jugabilidad"? ¿Acaso el que una persona no quiera dejar de jugar a nuestro juego, o no deje de pensar en el puzzle que le hemos puesto, no evidencia que esas sensaciones le llegan de la forma más directa posible? No es "ver a un piloto de carreras" es SER un piloto de carreras.No es "¿encajarán las piezas?" es ENCAJAR las piezas.No es "A ver cómo sale el bueno de ésta" es ¡¡¿Cómo SALGO de ésta?!! Sí, los videojuegos nos cuentan una historia, nos atrapan en su mundo con sus gráficos, su sonido, su ambientación... pero hay algo que ningún otro medio tiene, al menos en tan gran medida: la INTERACCIÓN. Y es que en aquellos videojuegos, sin importar el tipo de historia, o el género del juego... NOSOTROS éramos el protagonista. Y la historia podía terminar bien o mal, todo dependía de nosotros. No había un final pre-fijado, no éramos simples espectadores: vivíamos una aventura. Era JUGAR, con mayúsculas. ¿Cuántas veces en estos años nos hemos sentido un científico metido a Rambo, un aprendiz de pirata o de mago, un piloto de carreras, o incluso una gran boca amarilla que cuando se come unos puntos se encabrona y se zampa al primer fantasma que pilla? ¿Y acaso no hemos sentido alegría cuando hemos triunfado? ¿Tristeza cuando hemos fracasado? Júbilo, Odio, Rencor, Frustración, Humor, Amor... Cuando éramos "profanos", cuando no teníamos ni idea de lo que era un píxel, un polígono o un puntero, un juego era para nosotros, conscientemente o no, una fuente de EMOCIONES.

¿Por qué hacemos videojuegos? ¿De dónde sale ese impulso de CREAR? En el fondo todos sabemos la respuesta: queremos hacer sentir al resto de la gente las emociones que nosotros vivimos cuando jugamos a un videojuego. Porque, hora es ya de decirlo, la creación de videojuegos es un ARTE, y por lo tanto nosotros somos unos ARTISTAS. Y como artistas que somos llevamos a la gente, en forma de CD, emociones fantásticas y maravillosas. Y nuestra profesión por tanto, de programadores, diseñadores, músicos, grafistas, directores... de ARTISTAS al fin y al cabo, es transmitir esas emociones. No es, cómo muchos creen, "Soy un freaky que quiere demostrar que es el más listo". No es, cómo nosotros quisiéramos, "Voy a hacerme rico con este negocio tan moderno". Hacemos felices a los jugadores creando emociones para ellos. Lo que nosotros hacemos es algo MUY IMPORTANTE. Somos ARTISTAS, haciendo un ARTE. Un arte nuevo, que tiene un mercado que crece, que tiene un futuro espléndido, que muy poca gente es capaz de realizar...

Desde aquí mi más sincero agradecimiento a todos los creadores por haberme EMOCIONADO con sus juegos. El cine ha evolucionado mucho desde los hermanos Lumiére (1895, año que se inventó el cinematógrafo). Nuestro arte nació hace poco más de 30 años. Tenemos mucho camino por delante y será un placer recorrerlo juntos.

# APENDICE A

## Conceptos básicos de programación para el principiante

### *Lo primero es lo primero: ¿qué es "programar"?

Pulsamos una tecla y los datos bailan en la pantalla o la impresora empieza a trabajar el papel. Un torno moldea con destreza el trozo de madera. El brazo robótico de la cadena de montaje aplica los puntos de soldadura con precisión. La máquina de refrescos nos ofrece nuestra bebida favorita y además nos devuelve el cambio con exactitud... Detrás de todas estas acciones están los programadores que son personas que se han encargado de elaborar unos programas (unas "recetas" que especifican qué acciones se han de realizar y cómo) para cada máquina determinada. Es decir,los programas no son más que un conjunto de instrucciones ordenadas, entendibles por las máquinas, y que les permiten realizar tareas concretas como las enumeradas al principio, (y muchas más que vemos a nuestro alrededor).

Así pues, para el buen funcionamiento de los aparatos anteriores podemos deducir que entran en juego dos elementos fundamentales: el procesador y el programa. El procesador es la parte física: se compone de una gran cantidad de elementos electrónicos (transistores en su mayoría) miniaturizados y encapsulados en una pastilla llamada microchip. Hay de muchos tipos y su estudio es muy interesante, pero no entraremos en más detalles. Y luego está el programa. No se trata de algo físico: ya hemos dicho que está compuesto de órdenes que van a guiar al procesador en sus tareas de manipulación de la información. Es decir, podemos definir finalmente un programa como un conjunto de comandos que un procesador ejecuta siguiendo un orden determinado para lograr alcanzar un objetivo propuesto.

Como programadores que somos, escribiremos esas órdenes en un archivo, como si escribiéramos un documento de texto, y luego se lo daremos como alimento al procesador. En el momento de la verdad, cuando queramos "poner en marcha" el programa, esas instrucciones que escribimos una vez deberán llegar al procesador en forma de impulsos eléctricos, y el conjunto de señales que interprete nuestro procesador dará como resultado la ejecución de nuestro programa.

### *Los lenguajes de programación

Si el procesador recibe las órdenes en forma de impulsos eléctricos, ¿cómo se escriben los programas?, ¿cómo codifico esas señales?. En el nacimiento de la ciencia de la informática electrónica (hace unos cincuenta años), los programas debían escribirse en un sistema que representara directamente los estados de las señales eléctricas que entendía el procesador.

Y los únicos estados eléctricos que un procesador entiende son sólo dos: o le llega corriente o no le llega. Por lo que a cada una de estas señales se la denominó bit y los dos posibles estados que podía recibir el procesador se representarían con un "1" (pasa corriente) o con un "0" (no pasa corriente) .Es decir, que en los albores de la informática, los programadores escribían directamente una secuencia de señales eléctricas codificadas en 0 y 1 (bits) en un orden determinado para que la circuitería de la máquina pudiera entender lo que tenía que hacer. Esto es lo que se llama el código máquina. Así que un programa (ficticio) podría ser, por ejemplo:

10010111010101011001100101101010101010101011010101001101010101 10

Hay que decir, que aparte de ser tremendamente difícil escribir un programa en código máquina (y tremendamente fácil equivocarse), el código máquina válido para un procesador no lo es para otro, debido a su propia construcción electrónica interna. Por lo tanto, un mismo programa se tenía que codificar en tantos códigos máquina como en procesadores se quisiera que funcionara.

Además, a medida que los adelantos tecnológicos permitían diseñar ordenadores más complejos y con

funciones más amplias, es fácil ver que la tarea del programador se iba volviendo cada vez más dificultosa y lenta, ya que las combinaciones de bits podían llegar a ser increiblemente largas y difíciles.

El primer gran cambio lo aportó la llegada del lenguaje Assembler (o Ensamblador). Se trata de un lenguaje de programación que asocia una instrucción (o un comando, es lo mismo) con una determinada subsecuencia de ceros y unos que un procesador concreto entiende. Así, no hay que escribir tiras inmensas de 0 y 1: simplemente se escriben instrucciones predefinidas que son simples sustituciones de subtiras concretas.

Seguidamente os pongo un ejemplo de programa Assembler que escribe la palabra "Hola" en pantalla.

```
.MODEL SMALL
.CODE
Programa:
MOV AX,@DATA
MOV DS,AX
MOV DX,Offset Palabra
MOV AH,9
INT 21h
MOV AX,4C00h
INT 21h
.DATA
Palabra DB 'Hola$'
.STACK
END Programa
```

Examinando el ejemplo nos damos cuenta enseguida de dos cosas: la primera, de lo complicado que es, no sólo utilizar, sino entender estas líneas de código. La segunda, aunque quizá no lo apreciemos tan claramente al no conocer otros lenguajes, de lo largo que resulta un programa que tan sólo escribe la palabra "Hola" en pantalla.

Precisamente por la característica de representar directamente las instrucciones de código máquina que soporta el procesador,el lenguaje Ensamblador pertenece a la categoría de los lenguajes llamados de bajo nivel, es decir, muy enfocados, cercanos y directos a la máquina. Afortunadamente para los programadores, la cuestión de los lenguajes ha evolucionado mucho y podemos distrutar actualmente de los lenguajes de alto nivel, los cuales poseen unas instrucciones escritas usando una nomenclatura mucho más próxima al lenguaje humano (generalmente, similar al inglés). De esta manera, con los lenguajes de alto nivel, leer el código de un programa nos puede servir para saber, más o menos, qué es lo que hace el programa sin que tengamos que tener grandes conocimientos de programación.

Uno de los lenguajes de alto nivel más importante es sin duda el lenguaje C. El mismo programa antes descrito podría escribirse en lenguaje C de la siguiente manera:

```
#include <stdio.h>
void main (void){
printf("Hola");
}
```

La instrucción printf("Hola"); resulta mucho más comprensible que todo el código en Ensamblador. Y ahora sí que podemos ver la diferencia de tamaño en el código total del programa.

Más allá de la distinción entre lenguajes de alto o bajo nivel, existe una gran variedad de lenguajes de programación. La mayoría es de propósito general, esto es, sirven para dar solución a problemas de ámbitos muy dispares. Lo mismo se usan para crear (o hablando más técnicamente, "desarrollar") aplicaciones científicas que para videojuegos. De este tipo de lenguajes tenemos el lenguaje C (aunque su aprovechamiento máximo se produce en la programación de sistemas a bajo nivel), C++ (una evolución de C), C#, Java, VisualBasic, Python, etc.

Otros lenguajes son de propósito específico -como Bennu- y están muy bien optimizados para la resolución de problemas de un campo muy concreto del saber. De este tipo tenemos lenguajes orientados al tratamiento de las base de datos empresariales como Cobol; lenguajes orientados a la resolución de problemas matemáticos como Fortran o MatLab; lenguajes orientados a la generación dinámica de páginas web como Javascript o Php; lenguajes orientados a la creación de videojuegos como Blitz3D o el propio Bennu, etc,etc.

La inmensa mayoría de todos ellos (los de ámbito general y ámbito específico) forman parte de los lenguajes de alto nivel pues su sintaxis es muy próxima al lenguaje humano. Sin embargo, sigue siendo imprescindible

traducir estos lenguajes al lenguaje que entiende el procesador, el código máquina. Para ello necesitaremos unos programas especiales, los compiladores y/o intérpretes, de los cuales hablaremos enseguida.

## *Editores, intérpretes y compiladores

Si podemos utilizar los llamados lenguajes de alto nivel que antes comentábamos, es gracias a la aparición de los intérpretes y de los compiladores. Los programadores escriben sus programas siguiendo la sintaxis (las normas) de un lenguaje determinado, por ejemplo C, y guardan el programa en un archivo como si se tratara de un texto. De hecho, el programa lo podremos escribir con cualquier editor de texto que tengamos disponible, incluso el "Bloc de notas" de Windows. Este archivo de texto que contiene el código de nuestro programa es el que se denomina código fuente del programa. A continuación interviene el intérprete o el compilador (o los dos, como el caso de Bennu) para traducir ese código fuente en código máquina, y posibilitar la ejecución del programa por el procesador. Es decir, los intérpretes y los compiladores son programas especiales cuya tarea es traducir el texto que hemos escrito nosotros a la secuencia de 0 y 1 que entiende el procesador.

Existen diferencias fundamentales entre los intérpretes y los compiladores, que va a decidir si el programador se decanta por el uso de un lenguaje de programación interpretado o compilado. Se resumen en las siguientes tablas:

| ***LENGUAJES INTERPRETADOS*** | |
|---|---|
| Son leidos y traducidos por el intérprete instrucción por instrucción, a la misma vez que se está ejecutando el programa. Es decir, el intérprete envia las instrucciones traducidas al procesador a la misma vez que las va leyendo del código fuente. Y esto ocurre cada vez que se ejecuta el programa. | |
| ***Inconvenientes*** | ***Ventajas*** |
| Esta manera de funcionar los hace más lentos, y además, para su correcto funcionamiento tiene que existir un intérprete allí donde el código deba ejecutarse. | Si el usuario tiene instalado el intérprete para un lenguaje concreto y lo que le suministramos es el código fuente, no importa el tipo de procesador que esté utilizando ni el sistema operativo que tenga instalado. Un programa en Java (lenguaje pseudointerpretado) podrá funcionar en un Pentium IV con Windows o con Linux, o en un Motorola montado en un Apple con el sistema MacOS instalado. Si existe un intérprete de ese lenguaje para una plataforma hardware determinada y para un sistema operativo concreto, se puede ejecutar allí. |

| ***LENGUAJES COMPILADOS*** | |
|---|---|
| El compilador se encargará de traducir el código fuente en código ejecutable directamente para el procesador y generará un archivo separado incluyendo este código binario. En Windows esos archivos binarios suelen llevar la extensión .exe (ejecutables). Así, el usuario solamente tendrá que ejecutar este archivo que tiene las instrucciones ya traducidas al código máquina. | |
| ***Inconvenientes*** | ***Ventajas*** |
| Precisamente el hecho de generar el código final a ejecutar por el procesador es lo que trae su mayor desventaja. El código máquina se generará para un procesador de una familia determinada -hemos comentado antes que los códigos máquina son muy poco transportables- , por ejemplo para los Pentium, y no podrá ejecutarse en otra plataforma, como un Macintosh. Y no sólo para las plataformas hardware, sino también para los sistemas operativos. Un programa compilado para Linux no funcionará bajo Windows. Esto obliga a los programadores a recompilar el programa para las diferentes plataformas y entornos en los que va a trabajar, generando diferentes versiones del mismo programa. Además el programador también se encuentra con el trastorno de tener que recompilar el programa cada vez que le hace una modificación o corrección al código fuente. | El archivo binario obtenido está muy optimizado y su ejecución es muy rápida, ya que no requiere de más traducciones. Además, el usuario no necesita ningún programa adicional para poder ejecutarlo y así el programador puede distribuir sus productos con más facilidad. |

Lenguajes interpretados son: Php, Python, Tcl, Perl, Javascript,... Lenguajes compilados son: C, C++, Delphi, Cobol, Fortran... ¿Y Bennu qué es: compilado o ejecutado? Pues, al igual que Java ó C#, las dos cosas a la vez. Si quieres saber más, consulta el primer capítulo de este texto.

## *Las librerías

Cuando programamos en un lenguaje determinado, hacemos uso de una serie de instrucciones, órdenes o comandos (son sinónimos) que son los propios de ese lenguaje y que son, en definitiva, los que le dan cuerpo . Por ejemplo, para imprimir "Hola" por pantalla en C hemos visto que podemos hacer servir el comando *printf("Hola");*, en C# usaríamos el comando *Console.WriteLine("Hola");*, en Java el comando *System.out.println("Hola");*, etc. Cuando se diseñó cada uno de los lenguajes existentes, se dotó de un conjunto más o menos extenso de comandos para poder hacer algo con ellos: algo así como el vocabulario básico con el que poder trabajar con ese lenguaje.

La gente pronto se dió cuenta que con ese número limitado de comandos inicial era capaz de crear programas que, a su vez, podían ser utilizados como un comando más "no estándar" del lenguaje utilizado. Es decir, que existía la posibilidad de generar nuevos comandos para un lenguaje determinado, a partir de los comandos iniciales de ese mismo lenguaje, y hacer disponibles esos nuevos comandos para los demás programadores. La utilidad era obvia: ¿para qué escribir un programa de 100 líneas que me dibujara un círculo en la pantalla cada vez que quería dibujar el círculo ? Lo más inteligente era escribir ese programa una vez, y convertirlo en un comando más del lenguaje, para que así todo el mundo a partir de entonces no tuviera que empezar de 0 y escribir siempre esas 100 líneas, sino que solamente tuviera que escribir ese nuevo comando que él solito dibujaba un círculo.Así pues, cada programador se podía crear sus propios comandos personales que le facilitaban la faena: los codificaba una vez (es decir, escribía lo que quería que hiciera ese comando, a partir de comandos más elementales) y los podía utilizar a partir de entonces cuando quisiera. Era una forma perfecta para encapsular la dificultad de un programa y acelerar el desarrollo. Evidentemente, estos nuevos comandos - a partir de ahora los llamaremos funciones- se compartían entre los programadores, y a los que no habían diseñado esa función "no estándar", no les interesaba mucho cómo estaba hecho por dentro esa función, lo que les importaba es que hiciera bien lo que tenía que hacer.

Para poder compartir mejor entre los programadores estas nuevas funciones, pronto surgió la necesidad de agrupar las más populares o las más necesarias en conjuntos más o menos amplios: las librerías (o mejor, bibliotecas) de funciones. Las bibliotecas de funciones son, como su nombre indica, almacenes de funciones disponibles para un lenguaje concreto, que los programadores pueden hacer servir para facilitar tareas laboriosas que requerirían un montón de líneas de código si se partiera desde cero; gracias a que alguien previamente ya las ha codificado y las ha puesto disponibles en forma de función alojada en una biblioteca, se pueden evitar muchas horas de trabajo.

Para cada lenguaje hay disponibles bibliotecas de muchos tipos. Normalmente, las librerías viene agrupadas por utilidad: hay librerías de entrada/salida, que almacenan funciones que manejan el ratón y el teclado, hay librerías de sonido, que almacenan las funciones que gestionan la posibilidad de reproducir o grabar diferentes formatos de sonido, hay librerías GUI, que almacenan funciones que pintan ventanas botones automáticamente sin tener que empezar desde cero, hay librerías de red, que posibilitan tener funciones de comunicación entre ordenadores,etc. El nombre técnico para llamar cada conjunto de librerías en general es el de API (Application Programming Interface -Interfaz de programación de aplicaciones-), así que en nuestro programa podríamos incorporar una API de entrada/salida, una API GUI, una API de red,..., y hacer uso de las funciones incluídas en ellas.

De hecho, ya veremos que el lenguaje Bennu incorpora un mecanismo específico para poder utilizar librerías de todo tipo, escritas en C y/o Bennu, -normalmente, las APIs estarán disponibles para aquel lenguaje con el cual ellas mismas han sido creadas: Bennu se sostiene sobre C - dotando así al lenguaje de gran flexibilidad, extensibilidad y potencia, sin alterar para nada su núcleo central, el cual se mantiene pequeño y optimizado.

Para acabar, antes hemos hablado de que para programar necesitamos como mínimo un editor de textos, y un programa intérprete y/o compilador (depende del lenguaje). Pero estarás preguntando: las librerías, físicamente, ¿qué son?¿Dónde están? Normalmente las librerías son ficheros binarios -es decir, ya compilados- que para hacer uso de ellos has de tener copiados/instalados en tu sistema, bien en la misma carpeta que el archivo de texto con tu código fuente, bien en una localización estándar accesible. En Windows generalmente tienen la extensión .dll, y en Linux normalmente tienen extensión .so . Y evidentemente, si quiere usar las funciones que incluyen, tienes que hacer referencia a estos ficheros desde nuestro código fuente con un protocolo determinado según el lenguaje.

## *Librerías gráficas: Allegro, SDL, OpenGL, DirectX

Desengañémosnos: C++ (y C) es el lenguaje más utilizado para crear videojuegos a nivel profesional. Así que si queremos ser "profesionales", hay que conocer C++. C++ es un lenguaje muy potente y muy flexible, pero también bastante difícil de aprender. Además, C++ no incorpora funciones "nativas" (o sea, que pertenezcan al propio lenguaje en sí) para la gestión de gráficos ni multimedia, por lo que si nos decidiéramos por usar este lenguaje tendríamos que echar mano de alguna librería gráfica disponible y aprender a utilizarla.

Por librería gráfica se entiende aquella que te facilita mediante programación la tarea de dibujar en pantalla, ya sea bien en 2D o bien en 3D; y por multimedia que además dé soporte al control y gestión de sonido, música,vídeo,interacción teclado/ratón/joystick/...,etc.

Así pues, para incluir gráficos en los juegos y también incluir sonido y sistemas de control efectivos en C++ contamos con librerías pre-programadas. Si vamos a usar este lenguaje, sería una locura de nuestra parte y un suicidio profesional rechazarlas y hacer nuestros propios códigos (o rutinas, es lo mismo), desde cero: basta con aprender cómo funcionan las funciones incluídas en la librería que elijamos.

Nosotros no programaremos en C++ sino en Bennu, un lenguaje creado específicamente para hacer videojuegos (C++ es de propósito general) y infinitamente mucho más fácil de aprender. Evidentemente, hay algo a cambio de la facilidad tiene que tener Bennu: no es tan flexible como C++ y hay muchas cosas que no podemos hacer con él, pero ya hablamos de este tema en los correspondientes apartados del resto del presente libro.

A pesar de que programamos en Bennu, y en principio no tendríamos por qué saberlo, como programadores de videojuegos que seremos nos interesará conocer como mínimo de oído las librerías (o APIs) gráficas/multimedia más extendidas e importantes que existen para C++, y así algún día, quién sabe, dar el salto.

Existen cuatro librerías gráficas principales, dos de ellas consideradas "amateur" y otras dos consideradas "profesionales". Las primeras son Allegro y SDL, las segundas DirectX y OpenGL. Todas se pueden utilizar básicamente en C o C++:

| **Allegro** (http://www.liballeg.org) : |
|---|
| Excelente librería C multiplataforma (es decir, que permite generar código para Windows, Linux y otros sistemas sin modificar línea alguna) y de licencia libre (dominio público), diseñada especialmente para la programación de videojuegos, capaz de manejar gráficos vectoriales y bitmaps en 2 dimensiones (aunque en las últimas versiones permite además utilizar gráficos 3D gracias a la utilización de OpenGL), sonido, joystick, teclado, mouse y temporizadores. Al estar específicamente diseñada para la programación de videojuegos, todas las rutinas que utiliza están hechas para ser fáciles de manejar y sobretodo eficiente desde el punto de vista de un juego. Además de que tiene ya preconstruidas ciertas funciones para realizar algunos efectos que normalmente se tendrían que programar a mano.<br><br>En su página web se puede comprobar como Allegro dispone de múltiples bindings. Un binding es un "recubrimiento" de la API oficial de manera que ésta se pueda utilizar no sólo en el lenguaje para el cual fue concebida, sino para cualquier otro lenguaje de programación para el cual exista el binding adecuado. De esta manera, si existe por ejemplo un binding de Allegro para Php, un programador de este lenguaje podrá hacer uso de las funciones de Allegro sin ninguna necesidad de programar en C/C++. Esto permite que la librería amplíe sus posibilidades de ser usada por muchos más programadores.<br><br>Es una librería bastante simple de utilizar y suficientemente potente para una primera incursión en el mundo gráfico. No obstante, no es muy usada en juegos profesionales (o al menos no muchos aceptan que lo usan), quizás porque está diseñada con una visión general de cómo funciona un videojuego -por tanto, es ideal para novatos y como hobby- , pero en un juego profesional a los programadores les gusta exprimir hasta las raíces de lo que están utilizando. |
| **SDL -Simple DirectMedia Layer-** (http://www.libsdl.org) |
| Comparable a Allegro, es una librería C/C++ multiplataforma y de licencia libre (LGPL) diseñada para proveer acceso de bajo nivel a gráficos 2D, video, audio, acceso a red, teclado, ratón, joystick y además permite el uso de aceleración 3D via OpenGL Es muy recomendable para el que recién empieza porque es simple de programar y rápida. También tiene multitud de bindings para poder hacer uso de sus bondades en otro lenguaje de programación diferente de C, como por ejemplo C#, Java, Perl, Php, Python... |

El entorno Bennu (compilador, intérprete y librerías y módulos que lo integran) está programado en C utilizando la librería SDL. Por tanto, SDL es un pieza fundamental en la estabilidad, funcionalidad y potencia de Bennu.

Como ejemplo, aquí mostramos un código C que hace uso de SDL para mostrarnos en la barra de título de una ventana la frase ("Hola mundo!"):

```
#include <stdio.h>
#include <SDL/SDL.h>                        //Incluimos la librería SDL
int main (void){
     SDL_Surface *pantalla;                 //Definimos una superficie
     SDL_Event evento;                      //Definimos una variable de eventos
//Inicializamos SDL
     if( SDL_Init(SDL_INIT_VIDE) < 0 ) {  //En caso de error
          fprintf(stderr,"Error al inicializar SDL: %s\n", SDL_GetError());
          exit(1);
     }
     atexit(SDL_Quit);                      //Al salir, cierra SDL
//Establecemos el modo de pantalla
     pantalla=SDL_SetVideoMode(640,480,0, SDL_ANYFORMAT);
     if (pantalla==NULL) {
          fprintf(stderr,"Error al crear la superficie:%s\n", SDL_GetError());
          exit(1);
     }
     SDL_WM_SetCaption("Hola mundo!",NULL); //Escribimos el título de la ventana
     for(;;) {                               //Bucle infinito
             while(SDL_PollEvent(&evento)){
                      if(evento.type==SDL_QUIT) //Si es de salida
                               return 0;        //Se acaba el programa
             }
     }
}
```

Si quieres aprender más sobre la programación con SDL, recomiendo el curso http://softwarelibre.uca.es/wikijuegos/Portada .También te puedes descargar el libro "Programación de videojuegos con SDL para Windows y Linux" en formato Pdf desde esta dirección: http://www.agserrano.com/publi.html

**OpenGL -Open Graphics Library-** (http://www.opengl.org )

Allegro y SDL son librerías que por sí mismas sólo pueden manejar gráficos 2D. Si se quieren utilizar gráficos 3D, se requiere el uso de otra librería diferente.

OpenGL es una especificación oficial de una API 3D, controlada por un grupo llamado Architecture Review Board, conformado por varias empresas importantes del sector de las tarjetas gráficas como nVidia, ATI, Creative Labs, SGI, entre otras. OpenGL está pensado para funcionar en sistemas con aceleración hardware, por lo que los fabricantes de tarjetas han de incluir soporte OpenGL en sus productos.

A partir de esta especificación (que no es más que una serie de documentos), varias implementaciones pueden ser desarrolladas para obtener la librería 3D propiamente dicha. La implementación libre (básicamente con licencia MIT) y multiplataforma por excelencia de OpenGL se llama MESA (http://www.mesa3d.org). Ella misma está escrita en C, y las funciones que proporciona están pensadas para ser usadas en códigos escritos en C, pero si se quiere usar la potencia de OpenGL en otros lenguajes de programación, se puede consultar el listado de lenguajes disponibles en: http://www.opengl.og/resources/bindings .

OpenGL es la competencia de Direct3D, ya que, a diferencia de DirectX (en conjunto), esta API sólo posee rutinas para el manejo de gráficos y no tiene soporte para sonido, entrada, red, etc, el cual se ha de buscar en otras librerías (SDL, Allegro o más específicas).

**DirectX** ( http://msdn.microsoft.com/directx )

DirectX es una librería de Microsoft por lo que únicamente puedes hacer juegos para Windows con ella. DirectX fue diseñada por Microsoft con el fin de permitir a los programadores escribir programas multimedia para sistemas Windows de manera fácil, sin tener que preocuparse sobre qué hardware está corriendo y como funciona, como es el caso del tipo de tarjeta de video, etc, es decir, sin tener que escribir código específico de hardware. Para la mayoría de las aplicaciones tradicionales, el acceso a funciones de hardware se hacen a través del sistema operativo; sin embargo,

para las aplicaciones multimedia, que son más exigentes, esta vía puede resultar bastante lenta;DirectX es un conjunto de librerías que permitieran acceso a funciones de hardware de forma más directa, sin pasar por todos los callejones que usualmente crea el sistema operativo. Obviamente esto hace que , además de ser un producto de Microsoft, DirectX está íntimamente ligado con el sistema operativo Windows. Por eso, esto hace que todas aquellas aplicaciones sean, en extremo, difíciles de portar a otras plataformas,es decir, una vez que hayas terminado de programar tu juego estará condenado a trabajar solamente en Windows, y cuando quieras convertir tu juego para que pueda correr en otras computadoras que usen Mac o Linux tendrás que hacerle muchos cambio al código, lo cual no es nada deseable a menos que sepas que el único mercado al que va dirigido tu juego son personas con una computadora con Windows.

No obstante, una de las principales ventajas que tiene es que no está hecho solamente para la parte gráfica, sino que con DirectX puedes manejar toda la multimedia de la computadora como sonidos, música, dispositivos de Entrada/Salida como teclado, joystick, y varias otras cosas más, por lo que en este sentido es ideal para programa videojuegos. Realmente DirectX esta compuesto de varios componentes:

| |
|---|
| *Direct2D:* incluye toda la funcionalidad necesaria para la creación y manipulación de gráficos en 2D y otros servicios de video |
| *Direct3D*: es un motor de rendering para gráficos 3D en tiempo real que integra una API de bajo nivel para el render de polígonos y vértices y uno de alto nivel para la manipulaciión de escenas complejas en 3D. |
| *DirectInput*: permite recoger información en tiempo real del ratón, teclado, joysticks y más dispositivos de entrada |
| *DirectWrite:* es un renderizador de texto de alta calidad, independiente de la resolución y compatible con Unicode |
| *Window Sockets*: permite crear avanzadas aplicaciones de red capaces de transmitir datos online, independientemente del protocolo de red usado. |
| *Xact*: ofrece drivers y mezcladores de sonido con soporte Dolby, los cuales posibilitan un rendimiento óptimo de sonido posicional en 3D, permitiendo a los programadores de juegos situar eventos de sonido en cualquier punto del espacio perceptual del usuario, además de mezclar y manipular la reproducción de audio de forma avanzada. |

En el PC existe una gran variedad de tarjetas gráficas, de sonido, y demás hardware. Esto crea un problema a la hora de programar, ya que cada uno de stos dispositivos se programa de una manera diferente. DirectX, al igual que las otras librerías, es un puente entre el programador y la máquina. Dicho puente intenta evitar, en la medida de lo posible las diferencias que existen entre los distintos tipos de hardware que hay en el mercado.Por ejemplo, para hacer un sonido usaremos una instrucción de Xact, la cual será igual para todas las tarjetas de sonido, ya que es el propio Xact quien se encarga de dar las órdenes específicas a la tarjeta de sonido que tenemos conectada a nuestro PC. Si no existiera Xact, tendríamos que hacer rutinas de sonido diferentes para modelos de tarjetas de sonido que no fueran compatibles entre sí.

Por lo tanto, finalmente podríamos concluir, a modo de esquema, que en el panorama de la programación multimedia existe hoy en día podría resumirse en esta correspondencia:

| ***Librerías no portables*** | ***Librerías portables (y libres)*** |
|---|---|
| Direct2D, DirectInput | SDL, Allegro |
| Direct3D | OpenGL |
| DirectWrite | Pango (http://www.pango.org) |
| Window Sockets | SDL_Net, alguna otra librería de red. |
| Xact | OpenAL (http://www.openal.org) |

*(OpenAL es una librería libre para el manejo del audio tridimensional; tiene la capacidad de acceder directamente a funciones avanzadas de la tarjeta de sonido y controlar la llamada aceleración de audio.Pango es una librería renderizadora de textos complejos. Ver más adelante)*

Un detalle que salta a la vista es que Microsoft ofrece una solución totalmente integrada y completa para todos los aspectos que conllevan la programación de un videojuego, y que en el otro lado tenemos multitud de librerías que cubren partes del todo, teniendo que buscarnos la vida con varias de ellas; estas librerías están dispersas y son muy heterogéneas, y su uso por parte de los desarrolladores implica bastantes problemas a la hora del desarrollo En este sentido, es remarcable el proyecto libre Tao Framework (http://www.taoframework.com) , el cual intenta integrar en un

todo multimedia multiplataforma coherente -mejor dicho, basado en la plataforma NET/Mono- algunas de estas librerías independientes, como son SDL, OpenGL, OpenAL, y otras más específicas que detallaremos más adelante, como ODE (http://ode.org) -un simulador de dinámica de cuerpos rígidos-, DevIL (http://openil.sourceforge.net ) -una librería para el tratamiento de imágenes-, e incluso incorpora un lenguaje de scripting completo como es Lua (http://www.lua.org) , intentando ofrecer así una competencia global a DirectX.

## *Otras librerías útiles para programar videojuegos en C/C++

Cuando uno programa videojuegos en C ó C++, además de alguna librería gráfica de las mencionadas en el apartado uso, normalmente hace uso además (o no) de otras librerías más complementarias y específicas que facilitan la labor del desarrollo del programa en diversos aspectos. La filosofía final es intentar aprovechar el trabajo que han hecho otros antes para no tener que reinventar la rueda cada vez. Por librerías "complementarias" se entiende cualquier tipo de librería que se encarga de alguna tarea muy especializada útil, en nuestro caso, para la creación de videojuegos.

Como programadores noveles de Bennu, raramente haremos uso de estas librerías, ya que están pensadas para necesidades bastante concretas y de la mayoría (aunque hay alguna excepción) no existe un binding para poderlas utilizar, pero no está de más conocer su existencia, para poder apreciar la diversidad de opciones que podemos llegar a tener a la hora de conseguir nuestros propósitos, y para que conocer someramente sus características principales, de manera que en un futuro próximo podamos, si es necesario, sacarles todo el jugo que ofrecen.

A continuación se presenta un listado de diferentes librerías de diversos tipos y finalidades. Se introduce primero la función principal de cada tipo de librería y posteriormente se detallan algunas distintas librerías concretas disponibles.

| ***MOTORES 3D*** | |
|---|---|
| Los motores 3D son librerías que ofrecen al programador la posibilidad de incorporar al videojuego iluminación 3D, texturas (es decir, imágenes usadas como superficie para un objeto tridimiensional que da apariencia de un material concreto), modelos 3D (es decir, los objetos individuales propiamente dichos que serán usados en la escena) y animaciones 3D, junto con el renderizado de todos estos elementos, de una forma muchísimo más sencilla, versátil, probada, completa y potente que si el código necesario lo hubiera desarrollado el propio programador. | |
| **Irrlicht** (http://irrlicht.sourceforge.net) | Motor de renderizado 3D libre (Zlib) multiplataforma usable en C++ y lenguajes NET nativamente (además de múltiples bindings) que evita la dificultad de utilizar las capas inferiores de librerías gráficas como OpenGL ó DirectX (es compatible con ambos). Se pueden construir terrenos y mundos enteros con él, además de animación de caracteres y efectos de partículas y luces/sombras. Ofrece también una GUI en 2D y muchas más funcionalidades. |
| **Ogre** (http://www.ogre3d.org) | Motor de renderizado 3D libre (LGPL) multiplataforma orientado a escenas usable en C++ que evita la dificultad de utilizar las capas inferiores de librerías gráficas como OpenGL ó DirectX (es compatible con ambos). Se pueden construir terrenos y mundos enteros con él, además de animación de caracteres y efectos de partículas y luces/sombras y muchas más funcionalidades. |
| **Crystal Space 3D** (http://www.crystalspace3d.org) | Motor de renderizado 3D en tiempo real, libre y multiplataforma. Capaz de renderizar movimiento de hojas vegetales en terreno, animación de esqueletos con sombras dinámicas...y en general, ofrece similares funcionalidades que los motores anteriores |
| **Panda3D** (http://www.panda3d.org) | Motor de renderizado 3D libre (BSD) y multiplataforma usable en C++ y Python. Similar en funcionalidades a los motores anteriores. |
| **Genesis3D** (http://www.genesis3d.com) | Otro motor de renderizado 3D, pero en tiempo real, libre con |

| | |
|---|---|
| | restricciones y multiplataforma. |
| **Cube** (http://cubeengine.com) | Motor 3D especializado en videojuegos multijugador de tipo "First person shooter". Es decir, de disparos en primera persona |
| **XreaL** (http://xreal-project.net) | Motor 3D especializado en videojuegos multijugador de tipo "First person shooter". Es decir, de disparos en primera persona |
| **OpenSceneGraph** (http://www.openscenegraph.org) | Motor 3D libre (LGPL), escrito en C++ y OpenGL, y multiplataforma. Especializado en la generación de escenas en tiempo real, no necesariamente creadas para videojuegos sino para otras posibles aplicaciones (por ejemplo: realidad virtual, simulación visual, etc) |
| **OpenSG** (http://opensg.vrsource.org/trac) | Motor 3D libre (LGPL) y multiplataforma especializado en la generación de escenas en tiempo real, no necesariamente creadas para videojuegos sino para otras posibles aplicaciones (por ejemplo: realidad virtual, simulación visual, etc) |
| **Torque3D** (http://www.garagegames.com) | Motor renderizador 3D privativo escrito en C++ sólo para Windows que incluye además un editor de mundos, un editor de terrenos, un motor de física, OpenAL y un lenguaje de scripting propio (TorqueScript) para unir todos estos componentes juntos sin necesidad de programar a más bajo nivel. |
| **Delta3D** (http://www.delta3d.org) | No es un motor 3D propiamente dicho, sino un motor de juegos completo (LGPL y multiplataforma), el cual está formado por diferentes componentes como OpenSceneGraph (ya comentada), OpenDynamicsEngine (ver tabla siguiente), Character Animation Library -como su nombre indica, es una librería especializada en esqueletos 3D para animar a caracteres, multiplataforma,escrita en C++- (http://home.gna.org/cal3d) y OpenAL, los cuales integra de forma coherente y global en una única API. |

| ***MOTORES DE FÍSICA*** | |
|---|---|
| Los motores de física son librerías que ofrecen al programador de videojuegos la posibilidad de simular de forma muy real fenómenos físicos, en los que intervienen multitud de ecuaciones matemáticas muy complejas y multitud de variables físicas (velocidad, aceleración, masa, fuerza,etc), tales como colisiones de múltiples cuerpos, el movimiento de partículas y fluidos (humo, fuego, aire, agua, olas, movimiento de ropa...), la rotación o deformación de sólidos (derrapes de coches, movimiento de submarinos, catapultas, bicicletas, asteroides, insectos, explosiones, estructuras que colapsan), etc, de forma que el programador no se tenga que preocupar por las interioridades matemáticas de dichos procesos. Ejemplos de entornos que estas librerías pueden simular de forma fácil y rápida (y realista) para el programador pueden ser puentes, cuerdas, vehículos, un castillo de cartas, un brazo robótico, un humano, la suspensión de un muelle, etc. | |
| **Ode** (http://www.ode.org) | Librería libre (LGPL) y multiplataforma especializada en simular la dinámica del sólido rígido (por dinámica se entiende que el sólido se mueve o se deforma), detectar colisiones -con fricción-,calcular elasticidades... en 3D |
| **Bullet** (http://code.google.com/p/bullet) | Librería libre (MIT) y multiplataforma especializada en simular la dinámica del sólido rígido y blando, además de la detección de colisiones, deformaciones... en 3D |
| **Newton** (http://newtondynamics.com) | Librería libre y multiplataforma especializada en la simulación en tiempo real de entornos físicos (detección de colisiones, dinámica de sólidos, etc). |
| **Box2D** (http://www.box2d.org) | Librería libre y multiplataforma especializada en la simulación de colisiones, acoplamiento de momentos, gravedad...pero a diferencia de las anteriores librerías, trabaja en 2D |

| | |
|---|---|
| **TrueAxis** (http://www.trueaxis.com) | Librería para Windows de licencia no libre pero gratuita, especializada en detección de colisiones y simulación de vehículos, construcciones ensambladas y esqueletos |
| **Havok** (http://www.havok.com) | Librería para Windows privativa especializada en la detección de colisiones en tiempo real y la simulación de entornos físicos completos (esqueletos, vehículos,etc) en 3D. |
| **PAL**(http://www.adrianboeing.com/pal) | PAL no es una librería en sí, sino una API (un conjunto de funciones homogéneo) de acceso uniforme a múltiples motores de física, tales como Ode, Bullet, Newton, Box2D, Havok, etc. Es decir, su funcionalidad radica en ofrecer una manera siempre igual de acceder a la potencia de un determinado motor de física, independientemente de cuál sea éste en concreto. En su página web se muestra el listado de los motores de física con los que PAL es compatible; allí se pueden observar muchos más motores de física de los que se han listado aquí. |

| ***LIBRERIAS DE MANIPULACION DE IMAGENES/GRAFICOS*** | |
|---|---|
| No hay duda que un elemento fundamental en cualquier videojuego es el uso de múltiples gráficos, los cuales tendrán que ser cargados de disco,manipulados y mostrados en pantalla de la manera más conveniente. Incluso requerirán ser editados en tiempo real o se le tendrán que aplicar filtros para modificar su apariencia. Existen infinidad de formatos de imagen, y si el programador tuviera que crearse desde cero todas las rutinas necesarias para poder realizar todas estas tareas de edición (¡para cada uno de los formatos usados!), no acabaría nunca: es un trabajo demasiado largo y laborioso. Por eso existen una serie de librerías que facilitan mucho la labor al desarrollador, liberándolo de los problemas de bajo nivel para poderse concentrar en lo que realmente importa: la manipulación de los gráficos de forma sencilla y directa. | |
| **FreeImage** (http://freeimage.sourceforge.net ) | Librería libre (GPL) y multiplataforma capaz de cargar, salvar, convertir, manipular, mostrar y aplicar filtros a multitud de formatos de imagen, como por ejemplo .bmp, .ico, .jpg, .gif, .png, .tif, .psd...entre muchos otros, de forma sencilla para el desarrollador. |
| **DevIL** (http://openil.sourceforge.net) | Librería libre (LGPL) y multiplataforma capaz de cargar, salvar, convertir, manipular, mostrar y aplicar filtros a multitud de formatos de imagen, como por ejemplo .bmp, .ico, .jpg, .gif, .png, .tif, .psd...entre muchos otros, de forma sencilla para el desarrollador. |
| **MagickWand** (http://www.imagemagick.org) | API para C que da acceso a las librerías que conforman el conocido programa (libre (GPL) y multiplataforma) de edición de imágenes y aplicación de filtros ImageMagick |
| **SDL_image** (http://www.libsdl.org/projects/SDL_image) | Librería adicional a SDL (pero de la que Bennu no depende). Se encarga en concreto de cargar y manipular varios formatos de imagen sin tener el programador que lidiar él solo con rutinas de bajo nivel para estos menesteres.<br>Formatos soportados (entre unos pocos más) son:<br>-BMP<br>-XPM<br>-GIF<br>-JPEG (requiere tener instalada la librería JPG: http://www.ijg.org )<br>-TIF (requiere tener instalada la librería Tiff: ftp://ftp.sgi.com/graphics/tiff )<br>-PNG(requiere tener instalada la librería PNG: http://www.libpng.org/pub/png/libpng.html y Zlib: http://www.zlib.net)<br>Bennu sí que depende directamente de estas dos librerías, LibPng y Zlib, para poder soportar el formato Png nativamente. |
| **Cairo** (http://www.cairographics.org) | A diferencia de las anteriores, esta librería (libre (LGPL) y multiplataforma también) no está pensada para la manipulación de imágenes en mapa de bits sino para la creación y modificación de imágenes vectoriales en 2D (ver Apéndice B para más información). |

| | |
|---|---|
| | Esto la hace un poco diferente en su uso y sus posibles aplicaciones.<br>Otras librerías similares a Cairo serían Librsvg (http://librsvg.sourceforge.net) ó LibBoard (http://libboard.sourceforge.net) |
| **Ming** (http://www.libming.org) | La finalidad de esta librería escrita en C libre (LGPL) y multiplataforma es poder generar animaciones de tipo Flash (es decir, en formato vectorial SWF) "al vuelo". Otra librería similar (pero para C++ y con licencia MIT) es Sswf: http://www.m2osw.com/en/sswf.html |

| ***LIBRERIAS DE SONIDO*** | |
|---|---|
| Al igual que ocurre con las librerías de manipulación de gráficos, las cuales facilitan la labor al programador de videojuegos en el sentido de "enmascarar" los detalles más técnicos de su labor para poder centrar los esfuerzos en el desarrollo del propio videojuego y no en aspectos de más bajo nivel, con el sonido ocurre una cosa similar. Existen múltiples sistemas de sonido y múltiple hardware para lidiar con él y el programador de videojuegos no tiene porqué saber las interioridades de dichos sistemas. Las librerías de sonido vienen a completar este aspecto esencial en un videojuego como es el de la captura, gestión y direccionamiento de entrada/salida los flujos de sonido de forma fácil y accesible al programador. | |
| **OpenAL** (http://www.openal.org) | Librería libre y multiplataforma de audio 3D. Lo que hace es modelar una colección de fuentes de audio que se mueven en un espacio 3D, las cuales son escuchadas por un sujeto en algún lugar de ese espacio. |
| **SDL_mixer** (http://www.libsdl.org/projects/SDL_mixer) | Librería adicional a SDL (y en la que Bennu se basa también). Es un mezclador de audio multicanal. En concreto es capaz de soportar la reproducción simultánea de múltiples canales estéreo de 16bits más un canal de música, los cuales se mezclan mediante otras librerías de soporte a elegir, según el formato de audio:<br>-Para el formato Wav, Aiff y Voc no es necesaria ninguna librería suplementaria<br>-Para los formatos Mod, S3M, IT, XM (prácticamente obsoletos), se requiere la librería MikMod (http://mikmod.raphnet.net) , la cual ya viene incluida de forma integrada dentro de SDL_mixer.<br>-Para el formato Ogg Vorbis se requiere las librerías Libogg y Ligvorbis (http://www.xiph.org/downloads)<br>-Para el formato Mp3 se requiere la librería Smpeg (http://www.icculus.org/smpeg), la cual también es capaz de reproducir vídeos en formato Mpeg-1. Tanto Mp3 comod Mpeg-1 no está soportado nativamente por Bennu.<br>-Para Midi se requiere la librería (parcheada) Timidity (http://timidity.sourceforge.net) o hardware Midi nativo. Bennu no soporta Midi.<br>(Ver Apéndice B para más información) |
| **FMODEx** (http://www.fmod.org) | Librería de audio. Incorpora las funcionalidades que ofrece OpenAL además de las que también aporta SDL_Mixer, convirtiéndose pues en una librería ideal ya que es “todo en uno”. No obstante, no es libre ni gratuita, a excepción (lo de gratuita) cuando el proyecto que la utiliza es no comercial. |

| ***OTRAS LIBRERIAS INTERESANTES*** | |
|---|---|
| A la hora de programar videojuegos, debemos tener en cuenta muchos otros aspectos que van más allá de lo que sería estrictamente la programación multimedia (gráficos, sonido, 3D...). Un videojuego es un programa, y como tal, puede requerir de ciertas funcionalidades extra que pueden ser resueltas por diferentes librerías complementarias. A continuación se presenta una miscelánea de diferentes librerías de diversos ámbitos de aplicación que pueden ser útiles en un momento concreto para obtener una determinada funcionalidad. | |
| **AILoom** (http://ailoom.sourceforge.net) | Librería libre (LGPL) y multiplataforma escrita en C++ |

| | |
|---|---|
| | diseñada para facilitar la labor de programar elementos con inteligencia artificial dentro de un videojuego. Es bastante obsoleta, pero no se encuentran más alternativas (libres) debido a que en realidad el campo de la inteligencia artificial en los videojuegos es muy dispar: prácticamente cada videojuego necesita un motor propio de IA ya que los condicionantes de cada juego son muy diferentes y no permiten prácticamente la reutilización de un motor IA completo, así que es difícil obtener un motor IA genérico válido. Por motor IA se entiende un sistema que permite que elementos de un videojuego perciban su entorno y se comporten consecuentemente, tomando accions que maximicen sus probabilidades de éxito. Dentro del campo de la IA para juegos podemos encontrar aspectos como el seguimiento de caminos ("pathfinding"), los resolvedores de selección de acciones o el aprendizaje de la máquina, entre otros. Para más información, se puede consultar http://www.aigamedev.com .<br>Una alternativa (inacabada) a AILoom, también libre, podría ser OpenSkynet: http://openskynet.sourceforge.net<br>Un motor de IA (en 3D) privativo mucho más potente y profesional es Autodesk Kynapse.<br>Una librería libre escrita en C++ que solamente se encarga del pathfinding es MicroPather: http://www.grinninglizard.com/MicroPather |
| **OpenDBX** (http://www.linuxnetworks.de/doc/index.php/OpenDBX) | Librería libre (LGPL) y multiplataforma escrita C que permite el acceso a multitud de servidores de bases de datos (MySQL, PostgreSQL, MSSQL, Oracle, SQLite, Firebird,etc de una forma única, simple y limpia. De esta manera, el código escrito es prácticamente idéntico sea cual sea el gestor de bases de datos utilizado, permitiendo así una gran versatilidad y elegancia en el uso de diferentes sistemas gestores. |
| **LibDbi** (http://libdbi.sourceforge.net) | Otra capa de abstracción de acceso a base de datos, similar a OpenDBX. También está escrita en C, es libre y multiplataforma. Soporta menos sistemas gestores, no obstante. |
| **SDL_net** (http://www.libsdl.org/projects/SDL_net) | Librería adicional a SDL que proporciona soporte de red. Permite la transmisión de datos entre múltiples máquinas a partir del establecimiento de conexiones TCP y UDP , normalmente dentro de un esquema cliente-servidor. Bennu no la requiere. |
| **Boost Asio** (http://www.boost.org) | Otra librería libre C++ de red |
| **LibXML2** (http://www.xmlsoft.org) | Librería libre (MIT) escrita en C cuya funcionalidad es ser un "parser" (un analizador gramatical) de archivos XML, de forma que se puedan recorrer éstos obteniendo de ellos la información requerida, o bien escribir datos .<br>Otro parser escrito en C y también libre es Expat: http://www.libexpat.org |
| **TinyXML** (http://www.grinninglizard.com/tinyxml) | Librería libre (licencia Zlib) escrita en C++ cuya funcionalidad es ser un "parser" simple de archivos XML. |
| **SDL_ttf** (http://www.libsdl.org/projects/SDL_ttf) | Librería adicional a SDL que proporciona soporte para la manipulación y dibujado de fuentes TrueType en aplicaciones SDL. De hecho, no es más que un binding de la librería FreeType: http://www.freetype.org Bennu no la requiere. |

| | |
|---|---|
| **Pango** (http://www.pango.org) | Librería libre y multiplataforma capaz de renderizar textos complejos, administrando correctamente el uso de las fuentes y de la localización de idiomas. |
| **LibCwiid** (http://abstrakraft.org/cwiid/wiki/libcwiid) | Librería libre escrita en C que permite controlar el software que haga uso de ella mediante uno o más mandos de la consola Nintendo Wii (el "wiimote").<br>Otra librería similar es Wiimote-Api: http://code.google.com/p/wiimote-api<br>Otra más (esta vez escrita en C++) es Wiim: http://digitalretrograde.com/projects/wiim |

## *Lenguajes de programación especializados en la creación de videojuegos

Todo lo que hemos escrito en el apartado anterior respecto a las librerías nos sería útil en la práctica si programáramos videojuegos en C o C++. No obstante, hemos dicho que ambos lenguajes son bastante complejos y difíciles de aprender para un iniciado, con lo que fácilmente uno se podría desesperar y dejarlo todo correr.

Es por eso que existe Bennu. Un lenguaje de programación fácil y asequible para cualquiera. Con Bennu el programador no necesita introducirse en los intrincados mundos del C y de sus librerías: Bennu ya hace el "trabajo sucio" por él: así, el programador puede dedicar todas sus energías al diseño y programación de su videojuego, en vez de pelearse con el manejo de las tarjetas gráficas, la memoria y otros monstruos.

No obstante, como es natural, Bennu no es la única alternativa para desarrollar videojuegos de forma "casera". No es superfluo conocerlas por encima, para saber si nos podrían interesar, y conocer otras soluciones a los mismos objetivos.

Para empezar, tenemos el **XNA Game Studio** (http://msdn2.microsoft.com/es-es/xna/default.aspx), que es básicamente un intérprete y un conjunto de librerías asociadas, ofrecido por Microsoft gratuitamente -sólo en la versión Express- para el desarrollo específico de videojuegos. En realidad, XNA no es incorpora ningún lenguaje propio, sino que es un añadido al lenguaje C# para facilitar su uso en el cometido concreto de la creación de videojuegos, por lo que de hecho el desarrollador programará en C#, y por tanto, es evidente que para programar con esta plataforma se necesita tener instalado previamente de forma obligatoria un intérprete de C# , y es muy recomendable contar también con un IDE adecuado a este lenguaje. (Todo estos prerrequisitos se pueden encontrar -gratuitamente también- en http://msdn.microsoft.com/vstudio/express/downloads *)*.Finalmente, además también se necesita tener instalada la librería DirectX.

XNA está especialmente pensada para desarrolladores aficionados, estudiantes o independientes, y es una estrategia de Microsoft para crear una comunidad online de desarrolladores y juegos disponibles -previa suscripción en muchos casos- para Windows y también para su consola Xbox 360. Para saber más sobre esta plataforma, se puede consultar, a parte de la dirección anteriormente citada del XNA Developer Center, ésta otra: http://creators.xna.com o también http://www.xnadevelopment.com/index.shtml

En otro nivel, tenemos **3D GameStudio** (http://www.3dgamestudio.com) . Según lo que dice en su página web, "es la suite líder para creación de aplicaciones en tiempo real 2D y 3D. Combina la programación en lenguaje C-Script con un motor 3D de alto nivel, un motor 2D, un motor de dinámica física, un editor de modelos, un editor de niveles y un editor de terrenos, más librerías extendidas de objetos 3D, arte gráfico y modelos de juegos prefabricados". Según ellos nunca ha sido tan fácil crear juegos de 1ª persona (plano subjetivo) , de 3ª persona, RPGs (Rol Playing Games: es decir, juegos tipo "Zelda" para entendernos), juegos de plataforma, simuladores de vuelo, juegos de deportes, aplicaciones 3D, presentación en tiempo real,etc. Ofrece tres nivels de creación de juegos: con simples clics de ratón a partir de plantillas prediseñadas; programando en C-Script usando su compilador y depurador (programa que se dedica a detectar errores en el código fuente) o bien incluyendo el código C-Script a modo de bloque interno dentro de un código mayor programado en C++ o Delphi. Promete que incluso si ningún conocimiento de programación, siguiendo sus tutoriales paso a paso, un juego simple puede ser construido en una tarde, y usando el C-Script juegos de calidad comercial pueden ser creados y distribuidos con éxito. En la página web hay demos de juegos, sección de preguntas frecuentes (FAQs), un magazine de usuarios,etc.

Otra alternativa también a tener en cuenta es **BlitzMax** (http://blitzbasic.com) . Según lo que dice su página

web,"BlitzMax proporciona un entorno simple pero poderoso para la creación de videojuegos –simple porque se basa en el popular y cómodo lenguaje de programación BASIC, y poderoso gracias a un optimizado motor 2D/3 que trabaja en segundo plano. Incluye muchos comandos para ayudar al programador en la creación del juego, pero no demasiados: en vez de confundir con toneladas de comandos, el conjunto de éstos ha sido cuidadosamente diseñado para proveer la máxima flexibilidad con el mínimo esfuerzo. Blitzmax es la nueva generación del lenguaje de programación para videojuegos de la casa Blitz Research: mantiene las raíces del lenguaje BASIC de anteriores productos, como Blitz3D y BlitzPlus, pero añade un montón de nuevas características interesantes, como la mejora de su lenguaje BASIC (añade punteros a funciones, polimorfismo y herencia, arrays dinámicos, paso de parámetros por referencia, cadenas UTF16,soporte para conectar con código C/C++ y soporte para programar directamente en OpenGL,etc), soporte multiplataforma para Windows, Linux y MacOS, diseño modular en ficheros independientes, conjunto de comandos 2D extremadamente fáciles de usar, inclusión de un IDE – Integrated development environment- completo, etc. Y si el programador tiene problemas, siempre puede consultar la excelente documentación o acudir a la inmensa comunidad de usuarios en los diferentes foros de Blitz para pedir ayuda.". Un ejemplo de éstos últimos lo puedes encontrar en el estupendo foro de GameDevelopers http://portalxuri.dyndns.org/blitzbasico/index.php , donde podrás preguntar en castellano todas tus dudas sobre esta herramienta.

Otra alternativas (no están todas, ni mucho menos) que pueden ser tanto o más recomendables que las anteriores son:

| | |
|---|---|
| **PlayBasic** (http://underwaredesign.com) | Lenguaje para videojuegos 2D de estilo Basic |
| **DarkBasic** (http://www.darkbasic.com) | Similar al anterior pero permite trabajar también con 3D. |
| **Torque 2D** (http://www.garagegames.com) | Motor privativo de juegos en 2D multiplataforma, más IDE incorporado para programarlos en C++ o en un lenguaje propio, TorqueScript. Puede compilar juegos para Wii, Xbox o iPhone. Incluye también motor de física y librería de red y sonido. |
| **Clanlib** (http://www.clanlib.org) | Librería libre y multiplataforma para la programación de videojuegos 2D utilizando C++. |
| **SDLBasic** (http://sdlbasic.sourceforge.net/flatnuke) | Lenguaje interpretado basado en la librería SDL, multiplataforma y libre. Puede utilizar toda la potencia de SDL con la sintaxis Basic |
| **FreeBasic** (http://www.freebasic.net/index.php ) | Lenguaje de ámbito general de estiloBasic. Libre y multiplataforma. |
| **Xblite** (http://perso.orange.fr/xblite) | Lenguaje de ámbito general de estilo Basic pero con la velocidad de C. Libre; para Windows. |
| **PureBasic** (http://www.purebasic.com) | Lenguaje de ámbito general de estilo Basic, multiplataforma |
| **Real Basic** (http://www.realsoftware.com) | Lenguaje de ámbito general de estilo Basic, multiplataforma. |
| **Panda 3D** (http://www.panda3d.org) | Librería 3D para ser utilizada en programas escritos con el lenguaje Python (lenguaje libre y multiplataforma - http://www.python.org -). Vale la pena también mirarse http://www.pygame.org (permite utilizar la librería SDL con Python en vez de C) |
| **Crystal Space** (http://www.crystalspace3d.org) | Librería 3D libre para ser utilizada en programas escritos en C/C++ preferentemente para videojuegos. Incorpora también un motor propio. |
| **GDT** (http://gdt.sourceforge.net) | Librería libre para C/C++ enfocada en el creación de videojuegos, desarrollada por la comunidad hispana de los foros de Game Developers. |
| **Lua** (http://www.lua.org) | Lenguaje de script embebible, ligero y rápido, especialmente diseñado para el desarrollo de videojuegos y multimedia. Libre y multiplataforma. |
| **Basic4GL** (http://www.basic4gl.net) | Lenguaje Basic que incorpora un compilador con soporte para OpenGL. Permite así programar en esta librería 3D sin necesidad de utilitzar C/C++. Es libre pero sólo para Windows. |

| | |
|---|---|
| **FANG** (http://www.fangengine.org) | Paquete libre con clases Java que facilitan la programación en este lenguaje de videojuegos 2D con soporte de red. Especialmente pensado para estudiantes de programación |
| **Phrogram** (http://www.phrogram.com) | Anteriormente conocido como KPL -Kid´s Programming Language-, es un lenguaje de programación diseñado para ser fácil y divertido, y especialmente adecuado para la programación de multimedia/ videojuegos, más un IDE, con cierta similitud a Visual Basic. Se ha de ejecutar sobre una plataforma NET. |
| **Processing** (http://www.processing.org) | Lenguaje y entorno de programación libre destinado a la creación de imágenes, animaciones e interacciones. Especialmente pensado para artistas y diseñadores |

Por último, no me gustaría dejar de comentar al **Adventure Game Studio** (http://www.adventuregamestudio.co.uk/),el **RPGMaker** (http://www.enterbrain.co.jp/tkool/RPG_XP/eng)
o el **FPSCreator** (http://www.fpscreator.com) los cuales son programas especializados en generar aventuras gráficas el primero,RPGs el segundo y Shooters el tercero, sin prácticamente programación: a golpe de ratón.Pero claro, siempre llega un punto donde no son suficientemente completos ni tienen ni mucho menos la flexibilidad que un lenguaje de programación ofrece.

A destacar,no obstante, dentro de esta familia, el software generalista **GameMaker** (http://www.yoyogames.com/make) , en cuya web además se pueden descargar bastantes recursos como músicas, fondos y gráficos de diversa índole. Una alternativa libre a este último programa, todavía en desarrollo, es **Flexlay** (http://flexlay.berlios.de/ ).

Bueno, las diferentes propuestas parecen la panacea: ofrecen poder crear supervideojuegos, -3D además, cosa que Bennu de momento por sí solo no puede-, sin demasiado esfuerzo. Y probablemente sea verdad, pero la mayoría tienen un pequeño inconveniente (llamémoslo así). Aparte de que son productos comerciales y por tanto, para adquirirlos hay que pagar un precio, no son libres. Y eso, creo, que es un factor relativamente importante.

## Algunos recursos web más sobre programación de videojuegos

Finalmente, a continuación se listan enlaces a varios sitios web (algunos además incluyen foros de usuarios que te podrán echar una mano) que pueden ayudar a iniciarte en la programacion de juegos.

| | |
|---|---|
| http://www.vjuegos.org | Comunidad Iberoamericana de Desarrolladores de Videojuegos. Portal en español enfocado a impulsar el desarrollo de videojuegos a nivel profesional |
| http://www.gamasutra.com | El portal más importante sobre desarrollo de videojuegos a nivel profesional |
| http://www.gamedev.net | Excelente pagina para desarrollar videojuegos con cualquier librería. |
| http://www.devmaster.net | Web llena de recursos para programadores de videojuegos (foro -muy útil-, artículos, noticias, wiki, software...) |
| http://www.gdmag.com | Revista para Desarrolladores de Videojuegos |
| http://www.gdconf.com | Congreso Internacional de Desarrolladores de Videojuegos |
| http://www.igda.org | International Game Developers Association |
| http://www.adva.com.ar | Asociación de Desarrolladores de Videojuegos Argentina |
| http://nehe.gamedev.net | Página con tutoriales sobre OpenGL y más. |
| http://www.webgamebuilder.com | Portal donde podrás descargar múltiples herramientas (entre otras, el Blitz3D) para diseñar tus propios juegos, enfocados preferentemente a la web, y consultar los foros relacionados. |
| http://www.stratos-ad.com | Punto de encuentro para desarrolladores de videojuegos hispanos. |

| | |
|---|---|
| | Incluye bolsa de trabajo |
| http://www.codepixel.com | Artículos y tutoriales sobre programación gráfica y animación 3D. |
| http://www.ambrosine.com | Listado exhaustivo de enlaces a distintos programas creadores de juegos tipo GameMaker, diferentes lenguajes de programación de videojuegos, recursos como música, gráficos, fondos, etc listos para descargar |

Mención aparte requiere la estupenda web "The Game Programming Wiki" (http://www.gpwiki.org) , completísimo sitio donde se pueden encontrar artículos y referencias relacionados con los más dispares aspectos de la programación de videojuegos: desde exhaustivísimas y completísimas comparativas de diferentes lenguajes, librerías gráficas y motores gráficos , hasta tutoriales sobre planificación y diseño de proyectos; desde listados interminables de herramientas de diseño gráfico/creación multimedia e IDEs hasta especificaciones oficiales de formatos de archivos; desde formales artículos matemáticos que tratan sobre simulaciones físicas hasta direcciones legales de industrias del videojuego; desde entradas a diferentes portales de comunidades de desarrolladores hasta tutoriales específicos para Allegro, SDL, OpenGL, DirectX, DevIL, OpenAL, Java, C/C++, C#, Lua...; desde manuales de programación genéricos avanzados (estructuras dinámicas de datos, patrones de diseño,etc) hasta artículos sobre los diferentes métodos de Inteligencia Artificial o los algoritmos de encaminamiento tipo A*,etc,etc. IMPRESCINDIBLE.

## Sobre el "software libre"

Habrás leido en los apartados anteriores que existen diferentes librerías o lenguajes de videojuegos que son privativos (o lo que es lo mismo, no libres) y otros que sí son libres. Y que, en concreto, Bennu sí es libre (GPL). Pero, ¿qué significa realmente eso de "software libre"? ¿Por qué es tan importante que un programa sea o no libre? ¿Quiere decir que es gratis?¿Qué significa GPL,LGPL,BSD,etc?...En este apartado aclararé, por si no lo sabes todavía, lo que significa "software libre", y por qué es un concepto tan importante, no sólo para el mundo de la informática, sino para la vida en general.

Lo siguiente es un extracto de la definición de software libre obtenido de la web del Proyecto GNU (http://www.gnu.org ), el cual se encarga de fomentar el desarrollo de software libre a nivel mundial.

*"El "Software Libre" es un asunto de libertad, no de precio. "Software Libre" se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. De modo más preciso, se refiere a cuatro libertades de los usuarios del software:*

*-La libertad de usar el programa, con cualquier propósito (libertad 0).*
*-La libertad de estudiar cómo funciona el programa, y adaptarlo a tus necesidades (libertad 1). El acceso al código fuente es una condición previa para esto.*
*-La libertad de distribuir copias, con lo que puedes ayudar a tu vecino (libertad 2).*
*-La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. (libertad 3). El acceso al código fuente es un requisito previo para esto.*

*Un programa es software libre si los usuarios tienen todas estas libertades. Así pues, deberías tener la libertad de distribuir copias, sea con o sin modificaciones, sea gratis o cobrando una cantidad por la distribución, a cualquiera y a cualquier lugar. El ser libre de hacer esto significa (entre otras cosas) que no tienes que pedir o pagar permisos. También deberías tener la libertad de hacer modificaciones y utilizarlas de manera privada en tu trabajo u ocio, sin ni siquiera tener que anunciar que dichas modificaciones existen. Si publicas tus cambios, no tienes por qué avisar a nadie en particular, ni de ninguna manera en particular. La libertad para usar un programa significa la libertad para cualquier persona u organización de usarlo en cualquier tipo de sistema informático, para cualquier clase de trabajo, y sin tener obligación de comunicárselo al desarrollador o a alguna otra entidad específica.*

*La libertad de distribuir copias debe incluir tanto las formas binarias o ejecutables del programa como su código fuente, sean versiones modificadas o sin modificar (distribuir programas de modo ejecutable es necesario para que los sistemas operativos libres sean fáciles de instalar). Está bien si no hay manera de producir un binario o ejecutable de un programa concreto (ya que algunos lenguajes no tienen esta capacidad), pero debes tener la libertad de distribuir estos formatos si se encontrara o se desarrollara la manera de crearlos.*

*Para que las libertades de hacer modificaciones y de publicar versiones mejoradas tengan sentido, debes tener acceso al código fuente del programa. Por lo tanto, la posibilidad de acceder al código fuente es una condición necesaria para el software libre.*

*"Software libre" no significa "no comercial". Un programa libre debe estar disponible para uso comercial, desarrollo comercial y distribución comercial. El desarrollo comercial del software libre ha dejado de ser inusual; el software comercial libre es muy importante."*

Para que un programa sea considerado libre ha de someterse a algún tipo de licencia legal de distribución, entre las cuales se encuentran la licencia GPL (General Public License), o la LGPL, entre otras –hay muchas: el tema de las diferentes licencias es un poco complicado a nivel legal: son como los contratos de jugadores de fútbol: con muchas cláusulas; si quieres saber más, mírate http://www.opensource.org/licenses/category -. Estas dos licencias en concreto (GPL y LGPL) son genéricamente conocidas como licencias GNU y fueron redactadas en su día por la Free Software Fundation, fundación que se encarga de mantener con fuerza este movimiento, pero otras licencias libres importantes són la licencia tipo MIT, tipo BSD, tipo Mozilla, tipo Zlib, tipo Apache, etc. Una comparativa entre algunas de ellas la puedes encontrar en http://www.codinghorror.com/blog/archives/000833.html ó, de forma más completa, en http://es.wikipedia.org/wiki/Licencias_de_software_libre

Ejemplos de programas libres hay miles (en el Apéndice B se comentan varios), pero tal vez la estrella del software libre sea Linux. Por si todavía no lo conoces, Linux es un sistema operativo libre (GPL) y gratuito. Es decir, que en vez de que tu ordenador arranque en Windows, podría arrancar en Linux, accediendo de forma instantánea y gratuita a multitud de programas de todo tipo (ofimática, multimedia, juegos,etc). Si nunca has instalado Linux en tu ordenador y quieres ver cómo funciona, lo puedes descargar de multitud de sitios en Internet y probarlo. Efectivamente, multitud de sitios, porque como Linux es libre, ha habido gente que ha adaptado el Linux original a sus propias necesidad, de forma que actualmente hay muchas versiones –llamadas distribuciones- diferentes, aunque todas ellas son compatibles a nivel básico y más o menos semejantes. Algunas distribuciones importantes son: Ubuntu (http://www.ubuntu.com), Fedora (http://fedoraproject.org), Debian (http://www.debian.org), OpenSuse (http://www.opensuse.org) ó Mandriva (http://www.mandriva.com) , aunque si quieres ver más distribuciones y encontrar una que se ajuste más a tus necesidades, puedes visitar el portal http://www.distrowatch.com . Y lo mejor: en la gran mayoría de estas distribuciones no es necesario instalar nada en tu ordenador para poder probarlas: si no te atreves a instalarte tu Linux en casa porque temes destrozar tu Windows de toda la vida (que puede pasar si no vas con cuidado), las distribuciones mencionadas (y muchas otras más) funcionan sin más que metiendo el CD en la lectora y arrancando el ordenador. En ese momento habrás entrado en Linux de forma directa, listo para usar. En el momento de apagar, simplemente habrá que quitar el CD de la lectora. Al volver a arrancar, no habrá pasado nada: continuará Windows igual que estaba.

Después de todo lo comentado en este apartado, ya podemos hacernos cargo de lo que significa que Bennu sea libre (con licencia GPL, concretamente), y de lo que ello implica. Las consecuencias son, en pocas palabras, que cualquiera que quiera (y sepa, claro) puede ser parte partícipe del desarrollo de lenguaje. Es decir, tú mismo puedes contribuir a mejorar el lenguaje Bennu. Bennu no tiene dueño que diga cómo ha de ser. Todo el mundo es dueño de Bennu, y todos pueden hacer que evolucione como convenga. Esto va mucho más allá de si Bennu es gratis o no. Lo importante es que todos podemos acceder al código fuente de Bennu (es decir, Bennu no es más que un programa más: accediendo a su código (escrito en C + SDL) podemos ver cómo está hecho por dentro) y ver qué mejoras se pueden realizar o qué nuevas características se pueden implementar, y hacerlo, de tal manera que se cree una comunidad de personas que colaboren mutuamente para llevar a cabo todo este gran proceso altruista. Ésa es la gran diferencia de Bennu con los demás lenguajes de videojuegos: no es una caja secreta.

# APENDICE B

## Conceptos básicos de multimedia para el principiante

### *Gráficos (2D)

Seguro que tarde o temprano necesitarás imágenes para tu videojuego, y seguro que también necesitarás tratarlas (colorearlas, enfocarlas, ampliarlas, encuadrarlas,...) La primera pregunta es obvia: ¿de dónde podemos sacar las imágenes que necesito?

Si necesitas una imagen determinada puedes hacer una búsqueda en Internet, y si la tienes ya en la pantalla puedes hacer una impresión de toda o parte de ella para convertirla en una imagen (tanto en GNU/Linux como en Windows existen cientos de programas que permiten hacer eso: son los llamados "capturadores de pantalla"; aunque también puedes probar de utilizar la tecla "ImpPant", la cual coloca lo que se ve en pantalla en el portapapeles). Si dispones de la imagen impresa (en libros,revistas, fotografías…) puedes digitalizarla con un escáner. Si dispones de una cámara fotográfica digital puedes crear imágenes digitales a partir de la realidad. Si dispones de una cámara de video digital puedes extraer "fotogramas" o "encuadres" con el programa adecuado. Si tienes tarjeta capturadora de video puedes extraer fotogramas de cualquier grabación en video o de la señal de video de una cámara. Y siempre tienes el recurso de crear tú mismo la imagen con un programa de edición gráfica, ayudándote si quieres de algún periférico como una tableta gráfica, para poder dibujar mejor.

Una vez digitalizada, la imagen se convierte en un fichero. Dependiendo del tipo de fichero con que se guarde conservará características más o menos fieles a su fuente, tendrá más o menos calidad, será más o menos fácil retocarla sin perder calidad,será más o menos fácil de interpretar por cualquier ordenador…Todos estos factores vendrán determinados por 4 parámetros diferentes: la profundidad de color,la resolución, el peso (el tamaño que ocupa en el disco duro) del fichero de la imagen, i el formato de la imagen.

**Profundidad de color:**

La tarjeta de vídeo proporciona la comunicación necesaria entre el PC y el monitor. Recibe las señales digitales provenientes del procesador (el cual ha su vez las ha recibido del software que se esté ejecutando en el sistema en ese momento) y las convierte a un formato que pueda utilizar el monitor para crear una imagen visible. Más concretamente, la tarjeta de vídeo transmitirá al monitor la información necesaria para poder presentar en pantalla cada píxel de forma individual. Podemos definir un píxel como cada punto de luz que emite la pantalla; ésta está formada por multitud de ellos.

La profundidad de color se refiere a la cantidad de bits necesarios de la memoria que incorpora la tarjeta gráfica para que cada píxel pueda guardar toda la información relativa al color que muestra en cada momento. Por lo tanto, la cantidad de memoria que incorpora la tarjeta de vídeo es la que determina las profundidades de color que un sistema puede soportar (y también las resoluciones de pantalla, como ahora veremos).

Actualmente, gracias a la avanzada capacidad gráfica de las tarjetas, casi todas las imágenes son del tipo"true color" (color verdadero). Eso quiere decir que para cada píxel se necesitan 24 bits de memoria gráfica, o lo que es lo mismo, 3 bytes (8 bits corresponden a un byte), 8 bits para cada canal de color RGB "Red, Green, Blue" (o sea, Rojo,Verde y Azul, ya que la formación de color en la imagen se basa en la mezcla de estos tres colores primarios). La profundidad color verdadero proporciona una paleta de colores casi real para el ojo humano. No obstante, aunque cuanta más profundidad de colores tenga una imagen, mejor se verá (más información habrá para representar cada píxel), más espacio ocupará en la memoria de la tarjeta, y de rebote, más espacio ocupará el fichero en disco.

Los datos expuestos en el cuadro siguiente ayudarán a entenderlo mejor. El cálculo del número de colores se obtiene simplemente de contar cuántas combinaciones posibles diferentes de 0 y 1 se pueden obtener con un número de bits determinados. Por ejemplo, con dos bits existen cuatro combinaciones posibles: 00, 01, 10 y 11, y por tanto, cuatro colores posibles. Afortunadamente, existe una fórmula matemática para realizar el conteo de combinaciones de

una forma mucho más rápida que "a mano": Nºcolores= $2^{Nºbits}$

| *Número de Bits* | *Número de colores* |
|---|---|
| 1 | 2 (blanco y negro) |
| 2 | 4 |
| 4 | 16 |
| 8 | 256 |
| 16 | 65.536 |
| 24 | 16 millones |

**Resolución:**

La resolución define la cantidad de píxeles que contiene una imagen, en términos de anchura y altura de ésta (es decir: nºpixeles/unidadsuperficiepantalla). La resolución más corriente en los adaptadores de video fabricados actuamente es de 1024x768 pixeles, lo que significa que una imagen que ocupe toda la pantalla estaría formada por 30.720 píxeles. La relación más corriente entre el ancho y el alto de la imagen es de 4:3 de manera que los adaptadores que proporcionan una resolución mayor lo hacen utilizando la misma proporción. En general, utilizaremos una resolución de 640x480 píxeles en monitores de 14", 800x600 en monitores de 15", 1024x768 en monitores de 17", 1280x1024 con un monitor de 21", etc. Evidentemente, a más resolución, más pequeñas se ven las cosas (los píxeles son más pequeños ya que a mismo espacio hay más) y mayor es la sensibilidad y la calidad de la imagen.

La memoria de la tarjeta gráfica está formada por bits que están dispuesto –teóricamente- en tres dimensiones: la altura, que es el número de píxeles desde la parte superior a la parte inferior de la pantalla; la anchura, que es el número de bits desde la izquierda hasta la derecha de la pantalla, y la profundidad, que es el número de bits de memoria utilizados para cada píxel. Si una tarjeta utiliza por ejemplo un color de 4 bits, dedica cuatro bits de memoria para cada píxel. Esto quiere decir que cada píxel puede tener 16 colores –combinaciones- posibles. Para utilizar esta profundidad de color con una resolución de 640x480 la tarjeta tiene que tener 1.228.800 (640x480x4) bits, que equivalen a 150 kilobytes (KB) de memoria. Dada una tarjeta gráfica con una determinada memoria, sólo de podrán conseguir combinaciones de las tres dimensiones(anchura,altura,profundidad) que puedan ser admitidas por la memoria que tenga. Para los estándares actuales, una resolución de 640x480 con 16 colores es el mínimo admisible. Se conoce como resolución VGA estándar, y los sistemas la utilizan normalmente de forma predeterminada mientras no disponga de un controlador (un "driver") de video diseñado específicamente para la tarjeta instalada.

**Peso:**

Siempre que necesites reducir el peso de una imagen tendrás que renunciar a alguna característica de la imagen original: tendrás que prescindir de algunos datos (compresión con pérdida), o guardarla con menos colores de los que tiene en realidad (bajar la profundidad de color), o hacer que se componga de menos puntos de color que el original, y por tanto, que sea más "granulosa" (bajar la resolución) o hacerla más pequeña en tamaño (bajar los valores de las dimensiones de altura y anchura).

**Formato:**

El formato de la imagen es la manera como se codifica la información de ésta "físicamente" en el interior de un fichero. Hay dos grandes familias de imágenes según su formato: los gráficos vectoriales y los mapas de bits (también llamados gráficos rasterizados).

| *GRAFICOS VECTORIALES* | |
|---|---|
| Los ficheros con gráficos vectoriales contienen instrucciones (vectores) con información matemática sobre las posiciones, color, etc de las líneas y curvas que contienen las imágenes. Es decir, describen las imágenes a partir de sus características geométricas. La visualización de la imagen en la pantalla se basa en el cálculo de estos vectores y la consecuente reconstrucción, por parte del programa adecuado, de las formas y los colores. | |
| ***Inconvenientes*** | ***Ventajas*** |
| Cuanto más complejas son las imágenes, el ordenador | Son independientes de su tamaño de visualización, es |

| | |
|---|---|
| tarda más tiempo en calcularlas y en representarlas. | decir, se escalan automáticamente para aparecer nítidas (a eso se le llama no perder definición). Cuando editas o modificas la posición, forma, tamaño y color de un gráfico vectorial, éste no pierde calidad ya que lo que modificas son las propiedades de las líneas y curvas que lo definen. |
| No son aptos para mostrar fotografías o imágenes complejas. | Al estar definidas por vectores, las imágenes no pesan tanto en comparación con los mapas de bits, aunque esto depende mucho de la imagen y de la calidad que se desee: las imágenes formadas por colores planos o degradados sencillos son más factibles de ser vectorizadas. A menor información para crear la imagen, menor será el tamaño del fichero. |
| | Cada pieza de la imagen puede ser manipulada separadamente. Es posible mover objetos individuales alrededor de la pantalla, alargarlos, rotarlos, duplicarlos o introducir distorsiones,etc. |
| | Algunos formatos permiten animación. Está ser realiza de forma sencilla mediante operaciones básicas como traslación o rotación y no requiere un gran acopio de datos. |

Los formatos concretos de los archivos vectoriales (y sus correspondientes extensiones) pueden ser muy variados: PS y PDF (formato PostScript) ,SVG (el estándar internacional propuesto por el consorcio W3C) ,SWF (Flash), WMF (Windows MetaFiles), CDR,DFX, etc.

| ***MAPA DE BITS*** | |
|---|---|
| En contraste, se encuentran los gráficos formados por una cuadrícula de puntos de color (píxeles): los bitmap. Una imagen de este tipo está definida por la posición y por el color de cada uno de los píxeles que la configuran. Al modificar una imagen de mapa de bits se modifican los píxeles y no las línias o curvas. La representación del gráfico en pantalla es pues un mosaico formado por piezas de colores diferentes. | |
| ***Inconvenientes*** | ***Ventajas*** |
| Al depender del número de píxeles o resolución, estas imágenes pueden perder calidad al ser ampliadas (es decir, al escalarlas pierden definición). | Estas imágenes tiene la ventaja de reproducir rápidamente y con mucha fidelidad gradaciones sutiles de sombras y colores. Los bitmaps son típicamente usados para reproducir imágenes que contienen muchos detalles, sombras y colores: fotografías, negativos de películas y otras ilustraciones. |
| En comparación, pesan más que las imágenes vectoriales, ya que los bitmaps contienen información específica sobre cada píxel representado en la pantalla. | |

Los formatos concretos más corrientes de mapa de bits son PNG, GIF, JPG, BMP, PCX, TIF…Además, cada programa de retoque de imágenes suele tener un formato propio editable solamente por él mismo (por ejemplo, el Paint Shop Pro tiene el formato PSP, el PhotoShop el PSD, etc).

Comentemos algunos de los formatos de mapa de bits más importantes:

| | |
|---|---|
| **GIF** | *Se desarrolló específicamente para gráficos online, los cuales requieren tener poco peso para poder ser visualizados y descargados rápidamente.<br>*Utiliza un algoritmo de compresión sin ningún tipo de pérdida. El formato *.GIF consigue comprimir las imágenes rebajando el número de colores, o sea, rebajando la profundidad de color (es decir, la cantidad de información sobre la imagen que cada píxel necesita, expresada en bits). Como hemos comentado antes, cuanta más profundidad de color tenga una imagen más información sobre color tendrá que tener; por tanto, será de más calidad pero también ocupará más espacio en la memoria y el disco.<br>*Los gráficos están limitados a una paleta de 8 bits; es decir, 256 colores (el concepto de paleta se explica más |

| | |
|---|---|
| | abajo).<br>*Admiten transparencia. Eso quiere decir que tienen la posibilidad de convertir en transparente o invisible un solo color, de manera que el fondo que tenga ese color sea invisible.<br>*Permiten hacer animación con una técnica de poner muchas imágenes en el mismo archivo GIF. |
| **JPG** | *Se desarrolló como un medio para comprimir fotografías digitales.<br>*El algoritmo que utiliza descarta algunos bloques de datos de las imágenes cuando las comprime (utiliza pues una compresión con pérdida). Por esta razón es mejor sólo guardar una vez la imagen con compresión JPEG, porque cada vez que se utilice este algoritmo se eliminarán más datos.<br>*Permite escoger el grado de compresión: cuanta más compresión más pérdida y el peso del fichero es menor.<br>*Las imágenes en formato JPG pueden conservar una profundidad de color de millones de colores, reduciendo considerablemente el peso del fichero. Eso hace que este formato sea muy utilizado en Internet para facilitar las descargas, en las cámaras digitales (que tienen que poner cuantas más fotografías mejor en un espacio reducido), y,en general, siempre que se quiera trabajar con imágenes con muchas gradaciones de sombras y colores y se precise ahorrar espacio. |
| **PNG** | *Utiliza un algoritmo de compresión sin pérdidas, libre de patentes, que permite una mejora del 5-25 % mejor que el GIF.<br>*Que sea libre de patentes quiere decir que cualquier desarrollador puede diseñar programas que generen imágenes PNG, y utilizarlas sin ningún tipo de restricción. Esto no es así con otros formatos como el GIF: si un desarrollador quiere escribir un programa que en algún momento hace uso del algoritmo de creación de imágenes GIF (el algoritmo LZW), ha de pagar una suma económica por ello a la empresa poseedora de la patente de dicho algoritmo.<br>*No solamente el formato PNG está libre de patentes para su creación y uso, sino que es un formato libre. Es decir, la especificación del formato es pública y cualquiera puede modificarlo para mejorar el algoritmo de compresión de manera que toda la comunidad se beneficie de los avances en la evolución del formato.<br>*Una de las principales mejoras que incluye este formato es el denominado canal alfa que permite hasta 256 niveles de transparencia en el PNG-24 (24 bits), es decir que es posible tener transparencias con distintos grados.<br>*Con este formato no pueden realizarse animaciones aunque se ha desarrollado uno nuevo basado en este MNG (Multiple Network Graphics) que si lo permite. |
| **BMP** | *El formato gráfico estándar de mapa de bits sin compresión usado en el entorno Windows. Al no estar comprimido, las imágenes en este formato ocupan mucho más espacio en disco. |

Debido a ser un formato libre (y a que es de gran calidad), el formato PNG es el formato nativamente soportado por Bennu (a través de la librería LibPng). Esto quiere decir que Bennu está específicamente diseñado para trabajar con imágenes PNG, y por tanto, preferentemente todos los gráficos que se utilicen en un juego hecho en Bennu tendrían que ser PNG. Aunque es verdad Bennu soporta otros formatos,el recomendado sin duda es el PNG, por su integración y compatibilidad con la plataforma Bennu, su calidad intrínseca y su licencia libre.

**Otros conceptos:**

Por último, otros conceptos importantes que conviene aclarar en relación al tema de los gráficos 2D son los siguientes:

| ***CANALES RGB y ALPHA*** |
|---|
| Si un gráfico es de 24 bits, todos sus colores en la pantalla del ordenador se forman (ya lo hemos comentado antes) a partir de tres colores básicos, que son el rojo, el azul y –atención- el verde. El negro absoluto es la falta de color, y por tanto es aquel color que no contiene ni pizca de los tres colores básicos, es decir, el que tiene sus componentes RGB (Red-Green-Blue, o Rojo-Verde-Azul en español) a cero. El blanco entonces sería aquel que tiene cada uno de sus componentes RGB al máximo valor (porque el blanco es la suma de los colores primarios en plenitud). Cualquier color de entremedio tendrá una cierta cantidad de R, una cierta cantidad de G y otra de B, con lo que el número de combinaciones -y por tanto, de colores- posibles es mucho mayor a la que el ojo humano puede distinguir. No permite transparencias.<br><br>Si un gráfico es de 32 bits, al igual que ocurre con gráficos de 24 bits, se reservan 8 bits para cada color primario más 8 bits extra -llamados "canal alfa"- que sirven para especificar el grado de transparencia de ese color, desde totalmente transparente (valor mínimo de alfa, 0) hasta totalmente opaco (valor máximo de alfa, 255) en una graduación de 256 niveles distintos de transparencia. |

Si un gráfico es de 16 bits, se reservan 5 bits para el rojo, 6 para el verde y 5 para el azul. No permite transparencias.

Si un gráfico es de 8 bits, el sistema es diferente: disponemos de una (o más, pero sólo una activa en cada momento) paleta gráfica donde se pueden almacenar hasta 256 colores diferentes.

Comentario aparte son los gráficos con "escala de grises". En una imagen de escala de grises, cada punto es representado por un valor de brillo, que va desde el 0 (negro) hasta el 255 (blanco), con valores intermedios que representan diferentes niveles de gris. En esencia la diferencia entre una imagen en escala de grises y una imagen en RGB es el número de "canales de color": una imagen en escala de grises tiene uno; una imagen RGB tiene tres. Una imagen RGB puede ser pensada como tres imágenes en escala de grises superpuestas, una coloreada de rojo, otra de verde, y otra de azul.

***PALETA GRÁFICA***

Acabamos de decir que los gráficos de 8 bits (como por ejemplo, los que tienen el formato GIF) utilizan lo que se llama paleta gráfica. De hecho, la suelen incorporar como información extra dentro del propio fichero de la imagen, aunque también puede venir por separado. Utilizar una paleta gráfica significa que cada píxel de una imagen GIF (por ejemplo) tiene un valor numérico que es un índice dentro de esa paleta, el cual está asociado a un determinado color. Dicho de otra forma: la paleta es una simple tabla de color de empareja valores numéricos (los píxeles valdrán uno de ellos en un momento determinado) con colores (los píxeles mostrarán por pantalla el que se corresponda a su valor en ese momento).

Esa tabla de colores indexados se podrá construir según convenga, o bien distribuyendo en la paleta uniformemente unos colores seleccionados, disponibles a lo largo del espacio de color RGB, desde el violeta al rojo pasando por todos los colores intermedios (aunque hay que tener en cuenta que la cantidad de colores susceptibles de ser elegidos para incluirse en una paleta depende de la profundidad de color usada en ese momento), o bien llenando la paleta de gradaciones sutiles de un mismo color, y utilizar una u otra paleta allí donde sea necesario. Por ejemplo, si vamos a pintar un paisaje nevado, será más eficiente utilizar una paleta llena de colores con diferentes tonalidades de blanco, azul...Si vamos a pintar una selva, será mejor disponer de una paleta especializada en colores verdes, marrones...

Por ejemplo, en un modo de vídeo paletizado de 1 bit, cada píxel sólo podría tener dos valores posibles (0 ó 1), con lo que las imágenes que utilizaran este tipo de paletas sólo podrían ser en blanco y negro. Con una paleta de 4 bits, cada píxel podría tener un color dado por una de las 16 posiciones posibles en esa paleta. En un modo de vídeo paletizado de 8 bits, cada píxel sería un valor de entre 0 y 255. Por lo tanto, la paleta podría contener 256 colores y punto, que es lo que ocurre con las imágenes en formato GIF; la gracia está en crear una paleta de 256 colores que representen los colores que más usaremos.

***SPRITE***

Sprite es una palabra que se utiliza mucho en la literatura sobre videojuegos 2D, y conviene definirla para disipar dudas. Por sprite se entiende cualquier imagen 2D que se mueva en la pantalla, o más genéricamente, cualquier objeto 2D que puede ser visualizado por pantalla y que tiene asociado una serie de atributos como posición, velocidad, estado, etc.Así que un sprite puede ser el protagonista de nuestro juego, el ítem que recoge del suelo, un decorado, etc.

Los sprites en su origen fueron un tipo concreto de mapa de bits que se podían dibujar directamente por un hardware específico que liberaba de esta tarea al microprocesador (la CPU) del ordenador, reduciendo la carga del sistema. Por ejemplo, ese hardware específico se dedicaba (mediante operaciones AND y OR entre la imagen original y la del volcado, la cual se guardaba en una posición de memoria, lista para mostrarse en pantalla) al tratamiento de las transparencias sobre zonas de una imagen, entre otras cosas. Con el paso de los años el uso del término sprite se ha extendido a cualquier pequeño mapa de bits que se dibuje en pantalla sin tener en cuenta quien es el encargado de dibujarlo previamente

## *3D

Antes de nada, hay que aclarar que Bennu es un lenguaje que -a día de hoy- por sí mismo sólo permite programar juegos en 2D (es decir, si no hacemos uso de librerías externas, en cuyo caso sí que podremos utilizar Bennu en tres dimensiones), por lo que en este manual no tocaremos nada del mundo tridimensional. Sin embargo, en varios apartados del presente libro aparecen salpicados conceptos propios del mundillo 3D tales como aceleración 3D,

renderizado, etc, que es posible que desconozcas, por lo que es bueno explicarlos un poquito en este apartado para así tener las cosas más claras.

**¿Qué es animar?:**

La animación, ya sea 2D o 3D, se puede entender como una simulación de la realidad pero con unas leyes y unos objetos que le son propios. La animación es posible gracias a la llamada "persistencia retiniana", que es el efecto óptico por el cual las imágenes quedan durante unos instantes grabadas en nuestra memoria. Si hacemos pasar rápidamente diferentes imágenes ante nuestros ojos éstas se mezclan creando una sensación de movimiento. Para crear una buena sensación de movimiento hace falta pasar aproximadamente un mínimo de doce imágenes por segundo. A mayor número de imágenes por segundo la definición del movimiento mejorará. El cine como fotografía en movimiento necesita de un mayor número de imágenes por segundo, sobre todo en su grabación. Las primeras películas se rodaban a 16 o menos imágenes por segundo, de aquí el efecto estroboscópico de su movimiento. En la actualidad, tanto el cine como la televisión emiten y se graban a un mínimo de 24 imágenes por segundo ( o fps, frames por segundo).

La animación es el arte del movimiento: el movimiento parte de dos situaciones: A y B, y un desplazamiento con un tiempo determinado. Para ir de A hasta B necesitaremos un número determinado de pasos intermedios. Si queremos que el movimiento sea uniforme en la misma unidad de tiempo los espacios tendrán que ser iguales. Si lo que queremos es dar la sensación de aceleración o desaceleración tendremos que juntar o separar los espacios. La línea que marca la dirección del movimiento y sobre la cual marcaremos los pasos se denomina línea directriz. Además del desplazamiento sobre el plano, las figuras generan unos desplazamientos sobre ellas mismas, es decir, hace falta crear diferentes posiciones de la figura. Hace falta tener claro qué son las posiciones inicial y final que denominaremos posiciones clave (en dibujos animados se los denomina layouts, y en infografía keyframes). Estas posiciones nos muestran las características de lo que será el movimiento. El número de intervalos que queramos poner entre las dos posiciones principales dependerá de la velocidad de movimiento y la distancia, entre ellos del ritmo de aceleración o desaceleración que le queramos dar en cada momento.

Las animaciones 3D se ven actualmente sólo en 2D, pero su creación agrega realismo a las texturas, la iluminación, la profundidad de campo y otros muchos efectos que hacen que las imágenes parezcan más reales. La animación puede generarse, o bien en tiempo real, en el cual cada cuadro se crea con el tiempo del espectador o en tiempo simulado. Las animaciones de tiempo real son a menudo muy lentas puesto que la potencia informática necesaria para crear animaciones de alta calidad es sumamente elevada y, por lo tanto, muy caras. Actualmente, el ordenador genera todavía los cuadros, que se imprimen y fotografían o se envían a un dispositivo de salida de vídeo. De este modo, un ordenador puede estarse segundos, minutos, o horas generando cada cuadro, pero al reproducir la cinta o la película pasa cada cuadro en un fracción de segundo.

**Proceso de animación 3D:**

El proceso de animación por ordenador es una tarea compleja. En sus más amplios términos, la animación 3D se divide en las tareas de: guión y storyboard, modelaje, aplicación de texturas, animación, iluminación, sombreado y renderizado.

| | |
|---|---|
| **Guión y StoryBoard** | La creación de una película animada casi siempre empieza con la preparación de un "storyboard", una serie de bocetos que muestran las partes importantes de la historia y que incluyen una parte del diálogo. Se preparan bocetos adicionales por establecer los fondos, la decoración y la apariencia y temperamento de los personajes. En ciertas secuencias, la música y el diálogo pueden grabarse antes de ejecutar la animación final para que la secuencia final de imágenes sea gobernada por las pistas de sonido. |
| **Modelaje** | Es el proceso de crear y manipular una forma geométrica 3D para representar un objeto como en la vida real (o en la fantasía) en un ordenador; es decir, consiste en ir dando forma a objetos individuales que luego serán usados en la escena.<br>Se utilizan numerosas técnicas por conseguir este efecto que tienen coste y calidad diferentes: Constructive Solid Geometry, modelado con NURBS y modelado poligonal son algunos ejemplos. Básicamente, la técnica consiste en crear puntos en un "espacio virtual" que usa coordenadas X,Y,Z que representan anchura (X), altura (Y) y profundidad (Z) y conectar estos puntos mediante polígonos, curvas o splines -un tipo de línea especial-, según la calidad deseada.<br>Una alternativa a la creación de modelos propios, es emplear una gran colección de modelos ya creados, que está disponible en compañías especializadas en este campo. |

| | |
|---|---|
| **Aplicación de texturas** | Este proceso es el que contribuye principalmente a dar realismo a la visión del modelo acabado. No es suficiente con dar simplemente el mismo color al modelo que el objeto que se desea emular. Hay otros atributos que necesitan también ser considerados, como la reflexión, la transparencia, la luminosidad, la difusividad, el relieve, el mapeo de la imagen, el lustre, la suavidad y otros. |
| **Animación** | Es el arte de fabricar el movimiento de los modelos 3D. Hay muchas maneras de conseguirlo que varían en complejidad, coste y calidad.<br>Una manera muy cara y realista de animar a un ser humano es la "Captura del movimiento", para lo cual, se ponen sensores (captadores electrónicos de cambios de posición) en una persona y cuando ésta realiza los movimientos requeridos, un ordenador graba el movimiento a partir de las señales procedentes de los sensores. Los datos grabados pueden aplicarse después a un modelo 3D.<br>Otro método de producir una fluidez en los movimientos es una técnica denominada Cinemática Inversa, donde diferentes objetos (modelos) se unen jerárquicamente a otras formando como una cadena, y cuando uno de los objetos se mueve, los otras son influidos por esta fuerza y se mueven de acuerdo con los parámetros fijados, como si tuvieran un esqueleto. Con esta técnica, el animador sólo necesita poner sólo ciertas posiciones importantes conocidas como "keyframes" (cuadros clave) y el ordenador completa el resto. Con la ayuda de la técnica de keyframing, en lugar de tener que corregir la posición de uno objeto, su rotación o tamaño en cada cuadro de la animación, solo se necesita marcar algunos cuadros clavo (keyframes). Los cuadros entre keyframes son generados automáticamente, lo que se conoce como 'Interpolación'. |
| **Iluminación** | Es muy parecido a la de un estudio de televisión o de cine. Allá , según cómo y dónde se coloquen los diferentes tipos y cantidades de luz, queda alterada dramáticamente la percepción de la escena. La iluminación de la animación 3D utiliza los mismos principios y técnicas que en la vida real. |
| **Sombreado** | Son las diferencias de color de la superficie de un objeto debidas a las características de la citada superficie y de su iluminación. También afecta al sombreado la precisión con la que cada modelo particular es renderizado. Iría después de la iluminación. |
| **Renderizado** | Es el proceso de plasmar en una sola imagen o en una serie de imágenes todo el complejo proceso de cálculo matemático que se ha ido efectuado a partir del modelado, aplicación de texturas, animación, iluminación ,etc hecho hasta el momento por una escena concreta. Es decir, es convertir un modelo matemático tridimensional en una imagen (o video) de aspecto real, aplicando a cada superficie las adecuadas texturas, juegos de sombras y luces, etc... Dicho de otra manera, es el acto de "pintar" la escena final haciéndola visible, generada gracias a todos los cálculos necesarios; es obtener la forma definitiva de la animación a partir de todos los aspectos del proceso de animación 3D, dibujando el cuadro completo. Esto puede ser comparado a tomar una foto o lo caso de la animación, a filmar una escena de la vida real.<br>Hay muchas maneras de renderizar. Las técnicas van desde las más sencillas, como el render de alambre (wireframe rendering), pasando por el render basado en polígonos, hasta las técnicas más modernas como el Scanline Rendering, el Raytracing, la radiosidad o el Mapeado de fotones.<br>El proceso de render necesita una gran capacidad de cálculo, pues requiere simular gran cantidad de procesos físicos complejos, proceso que puede durar horas. El software de render puede simular efectos cinematográficos como el lens flare, la profundidad de campo, o el motion blur (desenfoque de movimiento) o efectos de origen natural, como la interacción de la luz con la atmósfera o el humo. Ejemplos de estas técnicas incluyen los sistemas de partículas que pueden simular lluvia, humo o fuego, el muestreo volumétrico para simular niebla, polvo y otros efectos atmosféricos, y las cáusticas para simular el efecto de la luz al atravesar superficies refractantes. |

**La aceleración hardware:**

Los videojuegos son una de las aplicaciones más exigentes, es por eso que a mediados de los años noventa, los desarrolladores se dieron cuenta de que los procesadores principales (CPUs) de los computadores personales no bastaban para los requerimientos de operaciones como el renderizado 3D. Fue entonces cuando a la

industria se le ocurrió que los PC deberían tener un procesador adicional, que se encargara exclusivamente de las operaciones relacionadas con los gráficos. A estos nuevos procesadores se les llamó GPU o Unidades de Procesamiento Gráfico; su objetivo es precisamente el de liberar a la CPU de las operaciones relacionadas con gráficos.

Las GPUs se colocaron en algunas tarjetas de vídeo, que fueron las que se conocieron popularmente como tarjetas “aceleradoras 3D”. De hecho, en aquellos años en el mercado se podían encontrar tarjetas de vídeo 2D "a secas" (hoy en día obsoletas), tarjetas aceleradores 3D "a secas" -que obviamente en un ordenador han de ir acompañadas por una tarjeta de vídeo de las anteriores - y tarjetas de vídeo con aceleradora incorporada, que proveían de cierto grado de aceleración por hardware. Hoy en día la inmensa mayoría de tarjetas de vídeo, en mayor o menor medida, incorporan aceleración 3D, por lo que uno de los factores de calidad a tener en cuenta en una tarjeta de vídeo actual es, además de la cantidad de memoria que aloja, la potencia de su GPU para realizar operaciones complejas.

La aceleración 3D “por hardware” se denomina así porque el renderizado tridimensional usa el procesador gráfico en su tarjeta de video en vez de ocupar valiosos recursos de la CPU para dibujar imagenes 3D. En la aceleración 3D “por software”, la CPU está obligada a dibujar todo por si misma usando bibliotecas de renderizado por software diseñadas para suplir esa carencia hardware,(como las bibliotecas de Mesa), lo que ocupa una considerable potencia de procesamiento. La aceleración tridimensional vía hardware es valiosa en situaciones que requieran renderizado de objetos 3D tales como juegos, CAD 3D y modelamiento.

Los gráficos 3D se han convertido en algo muy popular, particularmente en juegos de computadora, al punto que ya hemos visto que se han creado APIs especializadas para facilitar los procesos en todas las etapas de la generación de gráficos por computadora. Estas APIs han demostrado ser vitales para los constructores de hardware para gráficos por computadora, ya que proveen un camino al programador para acceder al hardware de manera abstracta, aprovechando las ventajas de tal o cual placa de video, sin tener que pasar por el molesto intermediario que representa el sistema operativo. Es por eso que una de las características de una tarjeta aceleradora es su compatibilidad con tal o cual librería gráfica que pretenda acceder a sus capacidades. Actualmente la mayoría de las tarjetas de vídeo toleran todas las librerías software nombradas en apartados anteriores (Direct3D,OpenGL,...) elegir se trata simplemente de optar por la que más nos convenga.

Pero, si en Bennu no se pueden hacer juegos 3D, ¿para qué necesitarás entonces las herramientas de modelaje 3D que comentaremos más adelante? La posibilidad más inmediata es para convertir una animación 3D en un video que se pueda mostrar en el juego a modo de intro, por ejemplo. De esta manera, via video, las animaciones 3D podrán incrementar el atractivo visual de tu juego en varios puntos.

## *Vídeo

**Procesamiento del video digital:**

Las 4 etapas básicas en el procesamiento del video digital son:

| | |
|---|---|
| **Captura/Digitalización** | El proceso de captura implica disponer de un ordenador con una tarjeta de captura. Su función es la conversión analógica-digital de la señal de video para la grabación al disco duro y posterior edición. La digitalización del video es un proceso que consiste en grabar la información en forma de un código de dígitos binarios a través del cual el ordenador procesa los datos electrónicamente. Una buena digitalización requiere una acción combinada de las prestaciones del ordenador y de la tarjeta digitalizadora (capturadora). La calidad se manifiesta en la velocidad de digitalización, el número de colores y las dimensiones de la imagen. De tarjetas capturadoras hay de todo tipo en el mercado: normalmente permiten entrar imágenes y sonido desde una cámara de vídeo analógica o digital, y exportar a un televisor, o al disco duro directamente, de manera conjunta o por separado. Es necesario, no obstante, hacer los correspondientes nexos a través de un cableado adecuado dependiendo del caso (RCA o S-Video para conexiones analógicas, Firewire –IEE1394- para conexiones digitales, etc). |
| **Edición** | Una vez realizada la digitalización, se dispone de herramientas informáticas que facilitan la edición (el tratamiento) de imagen y sonido y la generación de efectos: ordenar, editar con corte para desprenderse de los trozos no deseados, inclusión de transiciones, filtros, |

| | |
|---|---|
| | transparencias, títulos, créditos… |
| **Compresión** | Si se prevee difundir secuencias de vídeo, a las que son en formato digital se les ha de aplicar un proceso de compresión para reducir el tamaño de los ficheros. Existen diversos estándares de compresión, que se escogerán en función de la plataforma final de visualización. |
| **Difusión** | Una vez acabado el proceso de edición, es cuando se entra en la última fase : la exportación a un destino particular, que puede ser a una cinta de vídeo, un CD, un DVD, Internet… |

**Formatos de video digital (contenedores):**

Puedes pensar en los ficheros de vídeo como "ficheros AVI" ó "ficheros MP4". En realidad, "AVI" y "MP4" son sólo contenedores. Igual que un archivo ZIP puede contener cualquier tipo de fichero en su interior, los formatos contenedores de vídeo sólo definen cómo almacenar cosas dentro, no qué tipo de cosas tienen almacenadas (es algo más complicado que eso, porque no todos los «streams» de vídeo son compatibles con todos los contenedores, pero olvidemos eso de momento). Un fichero de vídeo habitualmente contiene múltiples pistas: una pista de vídeo (sin sonido), una o más pistas de sonido (sin vídeo), una o más pistas de subtítulos, y así sucesivamente. Normalmente las pistas están relacionadas; una pista de sonido contiene marcadores para ayudar a sincronizarse con el vídeo, y una pista de subtítulos contiene marcas de tiempo en las que mostrar cada frase. Cada pista individual puede contener metadatos, como la proporción visual en una pista de vídeo, o el idioma de una pista de sonido o de subtítulos. Los contenedores también pueden tener metadatos, como el título del vídeo en sí mismo, imágenes de la cubierta, números de episodio (para series de televisión), y cosas así.

Existen muchos formatos contenedores de vídeo. Entre los más populares están:

| | |
|---|---|
| **Audio Video Interleave** | El contenedor AVI fue inventado por Microsoft en tiempos más sencillos en los que el hecho de que los ordenadores fueran capaces de mostrar vídeos era algo sorprendente. No soporta oficialmente muchas de las características de los contenedores más recientes; no soporta oficialmente ningún tipo de metadatos de vídeo; ni siquiera soporta oficialmente la mayoría de los codecs de audio y vídeo que se usan hoy. A medida que pasaba el tiempo, varias compañías han intentado extenderlo de formas incompatibles para soportar esto o aquello. |
| **Advanced Systems Format (WMV)** | ASF fue inventado y es usado fundamentalmente por Microsoft en sus productos. Los ficheros suelen tener la extensión .asf ó .wmv. Los ficheros con sólo audio suelen tener la extensión .wma |
| **MP4** | Basado en un contenedor anterior desarrollado por Apple (QuickTime -MOV-), un contenedor anterior. Los ficheros suelen tener la extensión .mp4 ó .m4v, y son existen programas visualizadores para las dos plataformas (Mac y PC). |
| **Flash Video** | Utilizado fundamentalmente por Adobe Flash. Los ficheros suelen tener la extensión .flv |
| **Matroska** | Matroska , a diferencia de los anteriores, es un estándar abierto, no cubierto por ninguna patente conocida, y para la que hay implementaciones de referencia "Open Source" para hacer cualquier cosa que se desee con ficheros MKV. También existen los ficheros MKA (sólo audio) y MKS (sólo subtítulos). |
| **OGG** | Como Matroska, Ogg es un estándar abierto, amistoso al "Open source", y no cubierto por ninguna patente conocida. En los sistemas de escritorio, Ogg está soportado directamente por todas las distribuciones de Linux, y se puede usar en Mac y Windows mediante la instalación del software adecuado. |

Éste es el más breve de los resúmenes; hay literalmente docenas de contenedores en activo. DV, NUT, GP... la lista es interminable. Y hay muchos otros ensuciando el paisaje con tecnologías fallidas u obsoletas, procedentes de compañías que han intentado arrinconar el emergente mercado de vídeo definiendo sus propios formatos. Wikipedia tiene una buena comparación de contenedores en http://en.wikipedia.org/wiki/Comparison_of_container_formats

**Códecs:**

En la actualidad, el gran problema que tiene el video digital es el gran peso de los archivos generados,

aumentando éste todavía más cuando en el momento de la edición se añade sobre las imágenes un texto, un filtro o cualquier otro efecto especial.

Los datos siguientes ayudarán a entenderlo mejor: un fotograma de video con millones de colores, sin ningún tipo de compresión, ocupa alrededor de 1,2 Mbit. Como consecuencia, un segundo de video (25 fotogramas), sin comprimir, ocupa alrededor de 30Mbit y un minuto (1.500 fotogramas), 1.5 Gbit aproximadamente. Es evidente que se necesitaría un disco duro con mucha capacidad para poder almacenar todos los clips que se quieren editar, mucha memoria y un procesador muy rápido.

El problema de capacidad (y también el del volumen de datos en transmisiones) se resuelve mediante la compresión, conseguida gracias a la implementación en software (ó hardware ó una combinación de ambos) de diferentes algoritmos de compresión, llamados códecs, compresores o codificadores. La palabra códec viene de la contracción de las expresiones COder y DECoder. Así pues, las pistas de vídeo y audio incluidas dentro de un contenedor suelen ir comprimidas cada una con un códec distinto (de audio y de vídeo), y en aras a la reproducción de dichas pistas, será necesario descomprimir la información comprimida de forma adecuada. Es decir, explicando el proceso un poco más detenidamente, podemos considerar que cuando se crea un contenedor, en primer lugar se produce la codificación de las pistas mediante el códec elegido, y posteriormente son "unidas" (multiplexadas) siguiendo un patrón típico de cada formato. Cuando un archivo debe ser reproducido, en primer lugar actúa un divisor (splitter), el cual conoce el patrón del contenedor, y "separa" (desmultiplexa) las pistas de audio y vídeo. Una vez separadas, cada una de ellas es interpretada (descomprimida) por el descodificador y reproducida. Es pues imprescindible que el reproductor cuente con los descodificadores necesarios para reproducir tanto el vídeo como el audio, ya que de lo contrario la información no puede ser interpretada de forma correcta. En resumen, no sólo es necesario conocer el formato del contenedor para poder separar las pistas, sino que también es necesario poder descodificarlas.

La mayor parte de códecs provoca pérdidas de información para conseguir un tamaño lo más pequeño posible del archivo destino. Hay también códecs sin pérdidas (“lossless”), pero en la mayor parte de aplicaciones prácticas, para un aumento casi imperceptible de la calidad no merece la pena un aumento considerable del tamaño de los datos. La excepción es si los datos sufrirán otros tratamientos en el futuro. En este caso, una codificación repetida con pérdidas a la larga dañaría demasiado la calidad.

Los codificadores de vídeo más estandarizados actualmente son los siguientes:

| | |
|---|---|
| **M-JPEG** | Es una variedad del JPEG (Joint Photographic Experts Group), un sistema de compresión de fotografia que ha servido como modelo a muchos de los sistemas de compresión de video, entre los cuales hay el M-JPEG, el Indeo, el Cinepak, etc, que consideran el video como una sucesión de fotografías. Cuando comprime, digitaliza toda la información de cada fotograma de video, independientemente de los otros. Es lo que se conoce como Intraframe. Aunque ocupa mucho espacio, es el codificador que garantiza mayor calidad. Por esta razón, es el más usado cuando se ha de digitalizar material original que se ha de editar. El problema que presenta este sistema es la falta de compatibilidad entre las diferentes tarjetas digitalizadoras (capturadoras) existentes con el codec M-JPEG, ya que cada fabricante ha desarrollado su propia variante del formato. |
| **MPEG** | Es un sistema de compresión de video y audio implantado por la Unión Internacional de Telecomunicaciones.Al contrario que el anterior, es un codificador universal, no presenta imcompatibilidades. Comprime completamente diferentes fotogramas a la vez, y crea dependencia entre ellos. Es lo que se conoce como Interframe. Es un sistema más complejo. En el proceso de digitalización, comprime unos fotogramas principales, que son los de video y se llaman keyframes; y otros, secundarios, que contienen la información de las imágenes y se llaman deltaframes. Ahorra mucho espacio, pero no es muy utilizado para la edición. El hecho de no digitalizar completamente cada frame por separado comporta una pérdida de detalles no procesados, y consecuentemente una recogida de información incompleta que interfiere considerablemente en la calidad fianl, con imágenes poco definidas. En cambio, por su característica universal, es muy apropiado para ver videos ya editados.<br>Hay 3 tipos de MPEG, algunos ya se han comentado antes.<br><br>*MPEG 1: Diseñado para poder introducir video en un CD-ROM, con un calidad de imagen y sonido razonable.<br>*MPEG 2: Pensado para la televisión digital y para los DVD, por su alta resolución<br>*MPEG 4: Con un formato y una resolución baja para facilitar su transmisión, está ideado para videoconferencias e Internet, básicamente. |

| | |
|---|---|
| | Destacaremos el subformato MPEG-1 (Moving Picture Expert Group) que fue el estandar establecido en 1992. Se orienta al archivo y distribución de video digital. El MPEG-1 está pensado para un flujo de imágenes de 352x288 píxeles con 16 millones de colores a 1.5Mbits/s (calidad VHS). Posteriormente apareció el MPEG-2, pensado para la TV digital (cable y satélite) y es el formato utilizado en el DVD. Las pruebas actuales permiten distribuir secuencias de video de 720x485 píxeles. |
| **Theora** | Códec de vídeo (con pérdida) libre; es decir: sus especificaciones son públicas y libres de toda restricción legal para su uso y/o estudio. Está especialmente pensado para ser utilizado (generalmente junto con el códec de audio Ogg) dentro del contenedor Vorbis. De hecho, tanto Vorbis como Ogg y Theora han sido desarrollados por la misma fundación, Xiph: http://www.xiph.org Ofrece una alternativa libre y estándar a otros formatos privativos, como puede ser el MPEG-4. |
| **Dirac** | Códec de vídeo abiero y libre de patentes. Su objetivo es proporcionar compresión de vídeo de gran calidad , compitiendo con otros formatos como el H.264 ó el VC-1 |
| **XviD** | Códec de vídeo libre que sigue la especificación MPEG-4 ASP |
| **H.264** | Códec de video de alta calidad, también llamado MPEG-4 AVC. Incorpora diversas patentes que hacen que su uso sea desaconsejado. |

Algunos de estos codificadores ya están instalados por defecto en el sistema operativo del ordenador. Otros se instalan automáticamente cuando se instala la tarjeta capturadora de imagen y sonido, o bien algún programa de edición de video. Códecs importantes de este tipo son el Cinepak, el Sorenson, el Indeo…Hay que tener presente que éste es un mundo en contínua evolución, y que , por tanto, conviene irse actualizando e incorporando los nuevos códec que aparecen en el mercado según las necesidades.

La elección de un códec u otro depende del destino final de la película (DVD, Internet, videoconferencia, etc), el método de compresión que se hará servir y la forma como gestiona la imagen y el sonido. Si alguna vez tuvieras un video que no lo puedes visualizar correctamente debido presuntamente a que te falta el códec adecuado, para saber qué codec concreto es el que te falta y poderlo ir a buscar a Internet y descargarlo (un buen sitio para probar es buscarlo en http://www.free-codecs.com, donde entre muchas otras herramientas, se pueden descargar "Codec Packs" con multitud de códecs ya empaquetados en un solo instalador) puedes utilizar la estupenda herramienta Gspot (para Windows), descargable desde http://www.headbands.com/gspot .

Otra posibilidad es utilizar reproductores de vídeo que ya incorporan ellos de serie la mayoría de códecs necesarios. Algunos de ellos (todos multiplataforma) son: VLC (http://www.videolan.org/vlc) , Ffmpeg (http://ffmpeg.org) , Menplayer (http://www.mplayerhq.hu) o Miro (http://www.getmiro.com) .

Por si quieres profundizar en este tema, has de saber que a parte de cada página individual dedicada a un códec en particular (conteniendo multitud de información técnica sobre éste), Wikipedia tiene una buena comparación de códecs de vídeo en http://en.wikipedia.org/wiki/Comparison_of_video_codecs y http://en.wikipedia.org/wiki/Comparison_of_video_encoders . Otro link de interés es http://en.wikipedia.org/wiki/Open_source_codecs_and_containers y http://en.wikipedia.org/wiki/List_of_codecs

## *Sonido

El sonido es la sensación por la cual percibimos los cambios de presión y densidad del aire que se transmiten en forma de vibraciones. Tal y como hemos visto, los aparatos digitales pueden trabajar únicamente con secuencias de cifras numéricas. Igual que pasa cuando escaneamos una imagen, por ejemplo, el cual lee los colores de los puntos de la fotografía y los transforma en valores numéricos, si queremos trabajar el sonido con el ordenador nos hará falta obtener una representación numérica de estas vibraciones. Primero hace falta convertir las vibraciones del aire en oscilaciones de una corriente eléctrica. De esto se encargan los micrófonos y otros aparatos similares. El segundo paso consistirá a medir la intensidad de esta señal eléctrica a intervalos regulares. La colección de los valores obtenidos será ya una representación digital del sonido.

**El proceso de digitalización**:

En el proceso de digitalización intervienen tres factores:

| | |
|---|---|
| **Frecuencia de muestreo** | Indica el número de veces que se mide la intensidad de la señal eléctrica por segundo. Su unidad es el Hz (1 Hercio= 1 lectura por segundo).No se tiene que confundir esta magnitud con la frecuencia del sonido, donde los Hz indican el número de vibraciones por segundo. Cuanta mayor sea la frecuencia de muestreo, la captura de la señal será más fideligna al sonido real.<br><br>Los valores más usuales empleados en las grabaciones digitales son 11.025Hz para grabaciones de voz, 22.050Hz para grabaciones de música con calidad mediana y 44.100 Hz para grabaciones de música con alta calidad. |
| **Resolución de la lectura** | Siempre que se hace una medida hay un redondeo. No es el mismo pesar con unas balanzas de precisión, que nos permiten afinar hasta los miligramos, que con una báscula doméstica dónde siempre se acaba redondeando a decenas o centenares de gramos. Debido al sistema binario de numeración que usan los ordenadores, tenemos básicamente dos posibilidades:<br><br>-8 bits (un byte por lectura) Permite usar una escala de 256 valores posibles. Viene a ser como pesar con una báscula doméstica.<br>-16 bits (dos bytes por lectura) Ofrece una escala de 65.536 valores. Sería el equivalente a las lecturas que podamos obtener de una balanza de precisión. |
| **Número de canales** | La digitalización se puede hacer a partir de una señal monofónica (un solo registro sonoro) o estereofónico (dos registros simultáneos). |

**Digitalización de alta y baja calidad. Codecs:**

Los valores que escogemos para cada uno de estos tres parámetros determinarán la calidad de la digitalización, y nos indicarán también el número de bytes que necesitaremos para almacenar los datos recogidos. Los datos provenientes de una grabación digital de audio podrán ocupar mucho espacio, especialmente si la digitalización se ha realizado a alta calidad. Por ejemplo, para digitalizar una canción de 3 minutos de duración a 44.100 Hz se realizan casi 8 millones de lecturas:

*44.100(muestras por seg.) x 3(min) x 60(seg. cada minuto) = 7.938.000 muestras*

Si la grabación es estereofónico hará falta multiplicar por 2, y si las lecturas se hacen a 16 bits (es decir, calidad CD) necesitaremos 2 bytes por almacenar cada una de las cifras recogidas. En total la grabación ocupará:

*7.938.000(muestras) x 2(canales) x 2(bytes por muestra) = 31.752.000 bytes*

¡Es decir, casi 32 millones de bytes por sólo una canción!

Los códecs de audio (de forma similar a los códecs de vídeo) son algoritmos matemáticos que permiten comprimir los datos, haciendo que ocupen mucho menos espacio. Ya hemos dicho en el apartado anterior que en vuestro ordenador habrá instalados de serie unos cuantos códecs especializados en audio, y de otros especializados en vídeo.

Que usemos un códec "con pérdida" significa que al utilizarlo se pierde algo de calidad, puesto que se acostumbran a sacrificar los datos que nuestros sentidos no pueden percibir (por ejemplo, suprimiendo los armónicos más agudos que quedan fuera de las frecuencias audibles por los humanos). Por esto conviene hacer el proceso de compresión una sola vez, cuando ya hayamos realizado todas las modificaciones deseadas a los datos originales.

Los códecs de audio más usuales son:

| | |
|---|---|
| **MPEG-1 Layer 3** | También conocido como MP3 : Es el códec más extendido. Permite comprimir el sonido digital hasta 1/10 de su medida original sin que se pierda demasiada calidad. Se utiliza en muchos tipos de dispositivos portátiles, y es el rey de los códecs en el intercambio de música por Internet. |

| | |
|---|---|
| **Ogg Vorbis** | A diferencia de la MP3, que tiene un complejo sistema de patentes, este códec se basa en estándares de código abierto y libre, y la calidad es similar a de la MP3, si no mejor. El caso del códec Vorbis en el tema de licencias sería lo equivalente al formato PNG en gráficos: ya que el códec MP3 está patentado, su utilización para la creación de nuevo contenido puede estar limitada por las restricciones que imponga el propietario de la patente (actualmente, el Instituto Fraunhoffer), -básicamente el pago de royalties-, por lo que se vio la necesidad de crear otro códec de compresión de audio, con altas capacidades,que fuera libre, y gratuito. Que sea libre y gratuito quiere decir que los desarrolladores comerciales pueden escribir aplicaciones independientes que sean compatibles con la especificación Vorbis (y el contenedor Ogg) sin ninguna carga económica y sin restricciones de ningún tipo. Es evidente, otra vez, que este formato de audio es el preferido para ser usado con la plataforma Bennu Se soporta de forma nativa, con lo que la optimización está asegurada, y además su calidad es excepcional. |
| **AC3** | Códec multicanal (5:1) usado en muchos DVDs. |
| **GSM** | Es el códec empleado por los teléfonos móviles. Está pensado para comprimir el sonido del habla. Tiene una relación de compresión muy alta, pero la calidad obtenida es también muy limitada. |
| **AAC** | Advanced Audio Coding. Otro códec de compresión con pérdidas, patentado |
| **WMA** | Códec desarrollado por Microsoft, para ser utilizado con su contenedor ASF. |
| **Ogg Speex** | Códec libre para voz, sin restricciones de ninguna patente de software, usado con el contenedor Ogg. La diferencia con el códec Vorbis es que éste es más genérico para audio en general, y en cambio Speex está especialmente pensado para voz. |
| **Ogg FLAC** | Códec libre de compresión de audio SIN pérdida (a diferencia de los anteriores), usado con el contenedor Ogg. Otro códec sin pérdida sería el llamado Monkey´s Audio (APE). |
| **PCM** | Son las iniciales de "Pulse Code Modulation". De hecho el PCM no es un códec, sino el nombre que reciben los datos de audio digital sin comprimir. Lo incluimos en esta lista para ayudar a identificar los diversos formatos de codificación de datos. |

Wikipedia tiene una buena comparación de códecs de audio en http://en.wikipedia.org/wiki/Comparison_of_audio_codecs

Normalmente la extensión de un fichero de audio indica el códec que se ha utilizado en él (.mp3 para un fichero codificado en MP3,etc), pero hay casos en que no es tan fácil saber el códec presente en el fichero. Por ejemplo, la extensión .wav es la que se acostumbra a emplear en Windows para identificar los ficheros de audio digital, pero los datos de los ficheros .wav podan estar en formato PCM (sin comprimir) o pueden haber sido comprimidas con cualquiera de los códecs disponibles para Windows. Igualmente pasa con los ficheros de extensión .aiff (más comunes en sistemas Mac y Linux), los cuales suelen venir sin compresión pero no siempre.

**¿Qué es el MIDI?:**

El MIDI (acrónimo de Musical Instruments Digital Interface) es un sistema estándar de comunicación de información musical entre aparatos electrónicos. Para hacer posible esta comunicación, el MIDI describe dos cosas a la vez:

*Un método para codificar la música, basado en cifras numéricas que se transmiten una tras otra.
*Un sistema de conexiones físicas entre los aparatos:cables,conectores, tensiones eléctricas, frecuencias...

Es importante tener presente que el MIDI no sirve para transmitir sonidos, sino sólo información sobre cómo se tiene que reproducir una determinada pieza musical. Sería el equivalente informático a la partitura, al carrete de una pianola o al tambor de agujas de una caja de música: contiene toda la información necesaria para interpretar una pieza musical, pero si la queréis convertir en sonidos necesitaréis un buen instrumento y un músico o un mecanismo capaz de interpretarla. Así pues, hay que saber diferenciar los ficheros MIDI de los ficheros de sonido digital (WAV, MP3, Vorbis...). El MIDI es el equivalente a una partitura, y por esto sólo contiene las instrucciones que harán que un dispositivo compatible active los sonidos. El WAV, en cambio, contiene una descripción detallada de la ola de sonido resultante. Esto hace que la medida que ocupa un fichero MIDI sea muy menor a la de un fichero de sonido digital.

La ventaja de los ficheros de sonido digital es que ofrecen la misma calidad de sonido, independientemente del ordenador dónde los escuchamos. Por el contrario, la calidad del sonido de un fichero MIDI

dependerá del sintetizador o la tarjeta encargada de interpretar la "partitura". Por otra parte, una ventaja del MIDI respecto el sonido digital es su flexibilidad: la música se puede editar y modificar muy fácilmente, cambiando el tiempo, la alzada de las notas, los diferentes timbres utilizados, etc.

El sistema MIDI es empleado por varios tipos de aparatos musicales con funciones diversas. Podríamos agruparlos en tres grandes familias:

| | |
|---|---|
| **Aparatos generadores de sonido** | Son los aparatos que reciben información que proviene de algún otro y la transforman en sonido. Los aparatos MIDI pueden realizar esta función principalmente de dos maneras diferentes:<br><br>-Los sintetizadores generan el sonido de una manera totalmente artificial, basándose en combinaciones de funciones matemáticas por obtener los diferentes timbres. La tarjeta de sonido del ordenador incorpora un sintetizador MIDI.<br><br>-Los mostreadores (denominados también samplers) reproducen muestras grabadas de un instrumento tradicional. Las muestras de sonido se toman en un estudio y se almacenan digitalmente en la memoria del aparato MIDI. Posteriormente, son manipuladas por adaptarlas a diferentes niveles de intensidad y frecuencia. El sonido obtenido por este método puede tener una calidad parecida a la de una grabación en disco compacto hecho con el instrumento de dónde provienen las muestras. Algunas tarjetas de sonido de gama alta tienen, además del sintetizador, un sampler. |
| **Controladores** | Reciben este nombre los dispositivos especializados al emitir información MIDI. Podemos encontrar al mercado controladores que adoptan la forma de instrumentos convencionales (teclados, saxofones, flautas, guitarras, acordeones, baterías...) y de otras con diseños específicos (sensores de luz, de sonido o de movimiento, mesitas sensibles, platos giratorios...). |
| **Procesadores de datos** | Se especializan en recibir, almacenar, procesar y generar información MIDI. En este grupo encontramos los secuenciadores y los ordenadores. |

Actualmente todos los fabricantes de sintetizadores y aparatos musicales incorporan en sus equipos las conexiones y el circuitos atendiendo a la normativa estándar MIDI. Esto hace que siempre sea posible la comunicación entre los equipos, aunque no provengan del mismo fabricante o utilicen técnicas diferentes de síntesis y proceso de datos.

Algunos aparatos integran en una misma unidad elementos que pertenecen además de un grupo de los descritos. Por ejemplo, un sintetizador con teclado es a la vez un generador de sonido y un controlador, y puede traer también un secuenciador incorporado. En este caso la comunicación MIDI entre los diferentes componentes se efectúa internamente, sin cables ni conectores visibles, pero es importante entender qué de las funciones utilizamos en cada momento.

**Las tarjetas de sonido:**

En el proceso de grabación de un sonido, las señales analógicas de audio que llegan a la tarjeta, pasan primero por un amplificador que les proporciona un nivel adecuado para la digitalización. El amplificador amplifica más las entradas del micrófono que las entradas de línea, por lo que si se tiene una señal de una fuente que sea relativamente débil, se deberá conectar al puerto del micrófono en vez del de línea. Las tarjetas que funcionan en estéreo utilizan dos canales amplificadores. A continuación, el amplificador pasa la señal al convertidor analógico digital, y finalmente se pasa las señales digitalizadas al sistema principal para su almacenamiento o su uso en una aplicación.

En el proceso de reproducción de un sonido, las señales digitales de audio generadas por una aplicación que ejecuta un archivo de forma de ondas llegan a la tarjeta de sonido y pasan a través del mezclador. El mezclador combina las salidas de diferentes fuentes y entrega el resultado a otro amplificador que prepara la señal para su salida a los altavoces, auriculares o al sistema de sonido externo. Este proceso es similar a la reproducción de sonido via CD-Audio. Los programas reproductores son capaces de reproducir todos los formatos de fichero nombrados en apartados

anteriores, siempre que los códecs correspondientes estén instalados en el sistema. Hablando además bajo nivel, es la tarjeta de sonido la encargada de transformar la información digitalizada (de la manera que especifique el códec en cuestión) en un impulso eléctrico otra vez, el cual a través de unos altavoces se vuelve a convertir en sonido.

Cuando se utiliza una aplicación para reproducir un archivo midi, la tarjeta recibe la señal igual que antes, pero la información no se envía al mezclador directamente porque el midi contiene instrucciones de programación, no audio digitalizado. En su lugar, la información midi se envía al sintetizador, que genera el audio digital a partir de las instrucciones y pasa las señales de audio digital al mezclador, repitiéndose el proceso de antes.

## *Algunos programas útiles y recursos multimedia

Es evidente que para poder crear un juego mínimamente vistoso, debemos mostrar unos gráficos básicos: una nave, un asteroide, un fondo de estrellas, un disparo…Para poder incluir estos dibujos en nuestro programa, previamente deberemos crear los diferentes dibujos y sprites, e incluso si es necesario, crear nuestros sonidos para incluirlos también, animaciones 3D,etc. Para todo eso deberemos utilizar unas aplicaciones concretas fuera del entorno Bennu.

Se sale totalmente de los propósitos de este curso profundizar en el uso de las diferentes herramientas que un desarrollador de videojuegos necesitará para crear su mundo particular. Editores de imágenes, editores de vídeo, editores de sonido, suites 3D, etc son aplicaciones esenciales para poder generar un juego creíble, atractivo, interesante y profesional. Este curso sólo pretende iniciar al lector en el conocimiento de un lenguaje de programación concreto como es Bennu, dejando a un lado una explicación pormenorizada de todas estas herramientas que también son imprescindibles. En este sentido, el lector tendrá que hacer un trabajo de búsqueda y estudio particular de aquellas aplicaciones que en cada momento le convenga más o le ayuden mejor en su tarea de dibujar, crear músicas, incluir animaciones, etc.

Las posibilidades que se presentan al desarrollador son casi infinitas. Existen muchas aplicaciones para todo lo necesario, y en la elección de una herramienta particular influirá las necesidades particulares de un juego, el conocimiento multimedia del programador, la posibilidad de adquisión de la aplicación (algunas son libres) o incluso la apetencia personal.

A continuación ofrezco una lista incompleta, ni mucho menos rigurosa ni definitiva, de aquellas aplicaciones más importantes (preferentemente libres), interesantes de conocer. Dejo al lector la libertad de elegir la que considere más oportuna para sus propósitos, y repito que será tarea suya el aprendizaje del funcionamiento y posibilidades últimas de la aplicación escogida. Para un listado más exhaustivo, recomiendo visitar http://www.gpwiki.org/index.php/Tools:Content

| ***EDITORES DE GRAFICOS VECTORIALES*** | |
|---|---|
| **Inkscape** (http://www.inkscape.org ) | El editor libre y multiplataforma de SVG más completo y profesional. Totalmente recomendable por su sencillez de uso y su gran potencia y versatilidad. Las herramientas que incorpora son muy flexibles y útiles a la hora de trabajar con eficacia y rapidez. Viene provisto también de una serie de tutoriales perfectamente progresivos, estructurados y coherentes que facilitan en extremo el aprendizaje de esta herramienta |
| **OpenOffice Draw** (http://www.openoffice.org) | Dentro de la suite ofimática libre OpenOffice, aparte del procesador de textos, la hoja de cálculo, etc, nos encontramos con un simple pero muy funcional y eficaz diseñador de gráficos vectoriales. No se puede comparar en funcionalidad a los editores dedicados (OpenOffice Draw no deja de ser una parte más de una suite ofimática), pero para un usuario medio cumple de sobra su cometido. Para lo que haremos en este curso responderá de sobras a lo que podamos pedirle. |
| **Skencil** (http://www.skencil.org) | Programa de dibujo vectorial similar a CorelDraw o Illustrator, pero libre y multiplataforma.Programado en Python. |
| **Synfig** (http://www.synfig.org) | Software libre y multiplataforma orientado a la creación de dibujos y animaciones (sin interacción) en dos dimensiones. Especialmente pensado, |

| | |
|---|---|
| | pues, para creadores de "dibujos animados" clásicos. Trabaja con diversos formatos y tiene la posibilidad de utilizar papel calco, secuencias, sonidos, además de guardar en varios formatos de vídeo y en formato Flash. Otra alternativa libre es **Ktoon** (http://ktoon.toonka.com) pero parece discontinuada. Una alternativa comercial es **Anime Studio** (http://www.lostmarble.com) |
| **Flash** (http://www.adobe.com) | Aplicación comercial de referencia en lo que atañe al mundo multimedia en 2D. Creador de dibujos y animaciones tanto interactivas como no interactivas con inclusión de múltiples posibilidades multimedia, desde sonido y música hasta video. Es el programa original creador del formato SWF, el cual se utiliza en infinidad de webs para aumentar su atractivo visual. Incorpora además extensos tutoriales y documentación de referencia. Soporta además la posibilidad de programación de script mediante el cada vez más potente y flexible lenguaje ActionScript, transformando las últimas versiones de Flash en todo un entorno integrado para el diseño gráfico con la ayuda de la programación. No obstante,debido a sus posibilidades puede ser excesivamente complejo para el iniciado, además de no ser ni libre ni gratis. Algunas webs de soporte y ayuda con ejemplos de código ActionScript o animaciones son http://www.flash-es.net, http://www.flashkit.com o http://x-flash.org. Una aplicación similar, también comercial, es **Swish** (http://swishzone.com) |
| **FreeHand** (http://www.adobe.com) | Otro excelente editor (privativo y caro) de gráficos vectoriales de la casa Adobe, más centrado en el mundo editorial y de papel impreso. De esta misma casa también se puede obtener un programa muy similar, **Illustrator.** |
| **CorelDraw** (http://www.corel.com) | El equivalente al FreeHand, pero de la casa Corel. También es privativo (y caro) Una página de soporte a los usuarios de CorelDraw interesante es http://www.coreldev.org |
| **ArtRage** (http://ambientdesign.coml) | Programa de dibujo que simula el dibujo sobre un lienzo utilizando pinceles, lápices, tizas, etc. Se puede manejar con el ratón o con una tableta gráfica. |
| ***EDITORES DE MAPAS DE BITS*** | |
| **MSPaint** (de serie en Windows, aunque no libre) | Práctico, sencillo, rápido y sin opciones rebuscadas. Elemental e intuitivo. Y lo mejor, viene instalado de serie en Windows. Seguramente, será el editor de mapbits que usaremos más durante el curso, para hacer los bocetos de los dibujos que usaremos en nuestros juegos, sin necesidad de perfilarlos ni añadir detalles: para dibujar triángulos, cuadrados, líneas…De igual manera, su ayuda es concisa y clara: buena idea echarle un vistazo. |
| **TheGimp**(http://www.gimp.org) | La competencia libre al Adobe PhotoShop, la cual no le tiene nada que envidiar en prestaciones,potencia y calidad de resultados. Demostración palpable de que se puede desarrollar una aplicación libre que compita de igual a igual con los mejores programas propietarios de su rama. Altamente recomendable para editar las imágenes y fotografías con un resultado excelente. No obstante, al principio al usuario novel puede costarle entender el funcionamiento del programa, no es demasiado intuitivo. Atención, para poder hacerlo funcionar se necesita que previamente se haya instalado la librería gráfica GTK+ en el sistema. Dicha librería y la versión para Windows del Gimp las podrás encontrar si vas directamente a http://gimp-win.sourceforge.net .Una página web interesante de soporte al usuario en español, con muchos tutoriales, es http://www.gimp.org.es |
| **MtPaint** (http://mtpaint.sourceforge.net) | Sencillo programa de dibujo especialmente pensado para el Pixel Art -es decir, creación de dibujos mediante la edición de pixel a pixel- y la edición simple de fotografías. |
| | *Existen además aplicaciones muy interesantes llamadas trazadoras de imágenes, cuya función principal es convertir imágenes que son mapa de bits en imágenes vectoriales, para su manipulación posterior más cómoda. Un ejemplo de estas aplicaciones sería* ***Potrace*** *(http://potrace.sourceforge.net).* |
| **PhotoShop** (http://www.adobe.com) | El gran monstruo de la edición gráfica. El programa más potente, versátil y funcional. Con multitud de plugins que amplían el radio de acción de esta |

| | |
|---|---|
| | aplicación hasta el infinito. Con él se podrán hacer todos los retoques imaginables a fotografías y dibujos de forma totalmente profesional, además de aplicar filtros y efectos diversos. Posiblemente, tal cantidad de posibilidades pueda abrumar al usuario novel, ya que no es tan fácil de utilizar al principio como lo pueda ser un PaintShopPro o un Office Picture Manager, pero las ventajas que obtenemos con él no tienen comparación. Una web interesante de soporte con ayuda al usuario en español es http://www.solophotoshop.com |
| **Expression Studio** (http://www.microsoft.com/expression) | Suite de diseño gráfico de Microsoft formado por varias herramientas, entre las cuales se encuentra un editor de mapa de bits llamado Expression Design. |
| **PaintShopPro** (http://www.corel.com) | Excelente editor de dibujos y fotografías para el usuario medio. Ofrece las funcionalidades básicas que se le requieren a un programa de este tipo de una forma totalmente solvente,fácil y rápida. Dentro del paquete de instalación incorpora una utilidad muy práctica, **Animation Shop**, que es un generador de animaciones GIF. |
| **Office Picture Manager** (http://www.microsoft.com) | Similar en prestaciones al Paint Shop Pro: pensado para el usuario medio que no necesite de grandes prestaciones sino de las más elementales –a la par que más usadas-. Sencillo, rápido y eficaz. Este programa viene integrado como un paquete más a instalar en la suite ofimática Office. |
| **Fireworks** (http://www.adobe.com) | Aplicación pensada específicamente para la producción de elementos gráficos que se utilizarán en la web.Es decir, permite hacer dibujos, manipular fotografías y generar gráficos animados para Internet así como optimizar esas imágenes e incorporarles interactividad avanzada. |
| **Graphics Gale** (http://www.humanbalance.net) | Editor de gráficos y animaciones Gif muy fácil de utilizar, enfocado principalmente al PixelArt. |
| No obstante, si no quieres entretenerte creando tus propios dibujos/animaciones y deseas aprovechar las creaciones gráficas que existan ya realizadas por otras personas, en Internet hay multitud de webs de donde puedes descargarte gran cantidad de sprites, imágenes de fondo, iconos,etc gratuitamente para poderlos insertar en tus videojuegos inmediatamente. Puedes mirar por ejemplo:<br><br>http://www.gsarchives.net<br>http://www.panelmonkey.org<br>http://www.molotov.nu/?page=graphics<br>http://www.grsites.com<br>http://www.spritedatabase.net<br>http://www.sprites-inc.co.uk<br>http://www.sprite-town.tk<br>http://www.cvrpg.com/sprites.php<br>http://www.kingdom-hearts2.com/zerov/sprites.html<br>http://rcz.saphiria.net/html/sprites/sonic.php<br>http://www.nes-snes-sprites.com<br>http://www.goldensun-syndicate.net/sprites<br>http://dioxaz.free.fr/objets.htm<br>http://www.drshnaps.com/drshnaps/kirbyorigins/extras/sprites/ksprites.html<br>http://www.geocities.com/spritepage2003/update.html ,<br>http://lostgarden.com/2006/07/more-free-game-graphics.html<br>http://reinerstileset.4players.de/englisch.htm<br>http://forum.thegamecreators.com/?m=forum_view&t=85024&b=4<br>http://www.angelfire.com/sc/Shining/Metal.html<br>http://www.gamedev.net/community/forums/topic.asp?topic_id=272386 ,<br>http://local.wasp.uwa.edu.au/~pbourke/texture/ (texturas) | |
| ***SUITES DE ANIMACION 3D*** | |
| **Blender** (http://www.blender3d.org) | Blender es una suite integrada de modelación, animación, renderización, post-producción, creación interactiva y reanimación 3D-ideal para juegos-, con características tan interesantes como soporte a la programación bajo el lenguaje Python. Blender tiene una particular interfaz de usuario, que es implementada |

| | |
|---|---|
| | enteramente en OpenGL y diseñada pensando en la velocidad. Hoy en día todavía no exprime todas las posibilidades que los grandes productos ofrecen, pero éste es un proyecto que está gozando de una rápida evolución debido a la gran participación de la comunidad involucrada, así que es de esperar mejoras importantes en plazos cortos de tiempo. |
| **Art of Illusion** (http://www.artofillusion.org) | Art of Illusion es un estudio de modelado y renderizado 3D. Es un programa Java, por lo que se necesita tener instalado Java Runtime Environment 1.4 (o posterior). |
| **AutoQ3D** (http://autoq3d.sourceforge.net) | Programa de modelado 3D potente,flexible y fácil de utilizar |
| **Zmodeler** (http://www.zmodeler2.com) | Otro programa de modelado 3D libre y gratuito. |
| **Gmax** (http://www.turbosquid.com/gmax): | Software libre y gratuito de modelaje y animación 3D especialmente pensado para videojuegos. |
| **Anim8or** (http://www.anim8or.com) | Otro software libre y gratuito de modelaje y animación 3D. |
| **MilkShape3D** (http://chumbalum.swissquake.ch/ms3d/index.html) | Animador de esqueletos 3D. |
| También nos podemos encontrar programas más específicos que no abarcan todo el proceso entero sino que se dedican a modelar determinado tipo de imágenes. Por ejemplo, podemos utilizar un modelador libre de polígonos con mallas –sin animación- llamado **Wings 3D** (http://wings.sourceforge.net),o **Pov-Ray** (http://www.povray.org) , herramienta para crear imágenes tridimensionales fotorealistas mediante la técnica del trazado de rayos, o motores 3D listos para incorporar a proyectos de desarrollo, como **Irrlicht** (http://irrlich.sourceforge.net) , que es libre y multiplataforma, igual que **Ogre** (http://www.ogre3d.org) o **NeoEngine** (http://www.neoengine.org) . También nos podemos encontrar con generadores de explosiones, partículas y efectos especiales 3D, como **ExGen** (http://exgen.thegamecreators.com/) -de pago-, o **Particle Illusion** (http://www.wondertouch.com) -de pago también-. | |
| Algunas webs de soporte y ayuda referentes al tema del diseño gráfico 3D en general (sin asociarse a ningún producto en concreto pueden ser: http://www.3drender.com/ref, http://www.highend3d.com, http://www.beyond3d.com , http://humus.ca o http://www.es.3dup.com -ésta última en español- entre otros. | |
| **Maya** (http://www.autodesk.com/alias) | Es quizá el software más popular en la industria. Es utilizado por muchos de los estudios de efectos visuales más importantes en combinación con RenderMan, el motor de rénder fotorrealista de la empresa Pixar. Webs que pueden ayudar al usuario de esta aplicación pueden ser http://www.simplymaya.com y http://www.mayatraining.com |
| **3Dstudio Max** (http://www.discreet.com http://www.autodesk.com): | Es el líder en el desarrollo de 3D en la industria de juegos y usuarios hogareños. |
| **Lightwave 3D** (http://www.newtek.com) | Otro "monstruo" del sector. Como los anteriores, se compone de dos partes: modelador y renderizador. |
| **TrueSpace** (http://www.caligari.com) | Aplicación 3D integrada, con una apariencia visual muy intuitiva. Una característica distintiva de esta aplicación es que todas las fases de creación de gráficos 3D son realizadas dentro de un único programa, no como los demás que se componen de módulos separados. No es tan avanzado como los paquetes líderes Maya,3Dstudio,Ligtwave),pero provee características como simulación de fenómenos físicos (viento, gravedad, colisiones entre cuerpos) muy interesantes. |
| **Cinema 4D** (http://www.maxon.net) | Es un rápido motor de rénder. |
| **RealSoft 3D** (http://www.realsoft.fi) | Modelador 3D para Linux y Windows. Incluye rénder también. |
| **SoftImage XSI** (http://www.softimage.com/xsi) | Excelente software de modelado, renderizado y animación 3D enfocado a videojuegos y cine. |
| **DeleD** (http://www.delgine.com) | Modelador 3D enfocado en el diseño para videojuegos 3D.Tiene una versión Lite gratuita. |

| *EDITORES DE SONIDO* | |
|---|---|
| **Audacity** (http://audacity.sourceforge.net) | Editor de audio que permite reproducir, grabar en múltiples pistas, editar, mezclar, aplicar efectos, etc a archivos de formato de onda, de una manera cómoda y sencilla.Ideal para usuarios noveles o no tanto.Permite la utilización tanto de Wav como de Ogg, y Mp3 si se le incorpora el code LameMp3. |
| **WaveSurfer** (http://www.speech.kth.se/wavesurfer) | Herramienta para la visualización y manipulación de sonido en formato de onda, cuyas capacidades pueden ampliarse mediante plug-ins. |
| **Sweep** (http://www.metadecks.org/software/sweep) | Otro editor de sonidos en formato de onda (Wav y Ogg) |
| **ReZound** (http://rezound.sourceforge.net) | Y otro. Es parecido al CoolEdit. Permite aplicar filtros,efectos, loops, cue points,etc |
| **Mixere** (http://mixere.sourceforge.net) | Este programa libre no es un editor de sonido propiamente dicho: sólamente es un mezclador de ficheros de sonido. Muy sencillo pero perfecto para quien desee solamente esta funcionalidad a partir de ficheros ya editados Wav,Ogg,Mp3... para obtener curiosos efectos. |
| **Audition** (http://www.adobe.com) | Editor anteriormente llamado Cool Edit Pro. Ideal para manipular archivos de onda: incorpora múltiples efectos, filtros y posibilidades de edición de cualquier sonido Wav. Permite cambiar el formato del archivo y grabarlo en disco como Ogg.La herramienta completa para los usuarios medios. |
| **Nero Wave Editor** (http://www.nero.com) | Aplicación que viene integrada dentro de la suite de grabación Nero, la cual reúne todolo necesario para grabar y editar archivos de formato de onda de manera clara y directa. Las opciones que incorpora son las básicas, pero trabaja de forma muy optimizada. No obstante, por defecto no puede abrir ni grabar ficheros en formato Ogg, solamente Wav y otros (mp3,etc). |
| **Goldwave** (http://www.goldwave.com) | Otra excelente herramienta |
| A otro nivel muy diferente, ya completamente profesional, podríamos hablar de otro tipo de herramientas, no ya simples editores de sonido sino completas suites de composición,grabación y manipulación completa de música tanto MIDI como de formato de onda, como sería el **Protools** (http://www.digidesign.com ), el **CuBase VST** (http://www.steinberg.net ), el **SoundForge** (http://www.sonicfoundry.com ) o el **CakeWalk** (http://www.cakewalk.com ), las cuales son usadas en los estudios de grabación más importantes. Las alternativas libres a este tipo de programas todavía no están lo suficientemente evolucionadas para competir en condiciones de igualdad con las anteriores nombradas. Actualmente las más relevantes podrían ser los secuenciadores: **Rosegarden** (http://www.rosegardenmusic.org), el cual incorpora secuenciador MIDI y editor de partituras o, en menor medida **Ardour** (http://ardour.org).Ambos funcionan sólo en Linux. | |
| También podríamos hablar en este apartado de otro tipo de programas, que no son propiamente editores de audio, sino que lo que permiten es extraer -"ripear"- sonidos y músicas de CDs de audio y convertirlos en ficheros de formato de onda para, por ejemplo, editarlos y poderlos utilizar (siempre respetando la legalidad sobre los derechos de autor) en tus propios juegos. Por ejemplo, tenemos como aplicaciones libres y/o gratuitas: | |
| | **AudioGrabber** (http://www.audiograbber.com-us.net): Extrae canciones de los CD y los copia en formato de archivo Wav al disco duro en formato Wav y Ogg, entre otros. |
| | **Cdex** (http://cdexos.sourceforge.net ): Extrae canciones de los CD y los copia en formato de archivo Wav y Ogg al disco duro,entre otros. |
| | **BeSweet** (http://dspguru.doom9.net/) : Este programa libre no extrae canciones de CD: simplemente transforma el formato de los archivos de sonido seleccionados en otro. Soporta los formatos MP3,AC3,WAV,MP2,AVI,Aiff,VOB y Ogg. |
| | **Exact Audio Copy** (http://exactaudiocopy.de) : Extrae canciones de los CD y los copia en formato de archivo Wav y Ogg al disco duro, entre otros. |

| | |
|---|---|
| Respecto a los MIDIS, no podemos hablar propiamente de editores de sonido, sino de programas creadores y editores de MIDIS. Algunos podrían ser: | |
| | **Band in a Box** (http://www.pgmusic.com) ; Atención: este programa NO es libre. Es un generador de canciones. Aplicación ideal para componer melodías con multitud de arreglos armónicos yrítmicos. No obstante, se ha de saber un mínimo de solfeo y armonía musical para poderlo utilizar. |
| | **Visual Music** (http://shitalshah.com/vmusic): Permite tocar hasta 128 instrumentos musicales y grabar el resultado. También puedes escribir scripts para crear tus propias composiciones. Guarda el resultado en ficheros MIDI. |
| | **Jazz++** (http://jazzplusplus.sourceforge.net): Secuenciador de audio que permite grabar, editar y reproducir archivos MIDI. |
| | `Hydrogen` (http://hydrogen-music.org): Sintetizador de batería por software que se puede usar por sí solo, emulando una caja de ritmo basada en patrones, o via un teclado/secuenciador MIDI externo por software.Otro sotfware libre similar es **HammerHead** (http://www.threechords.com/hammerhead ). |
| Respecto la existencia de conversores de ficheros MIDI a Wav y viceversa se ha de decir que es escasa, debido a la dificultad que entraña la diferencia de estructura que poseen ambos formatos: uno está formado por instrucciones y otro por sonido digitalizado. Así que la transformación de uno a otro es muy costosa y no siempre con resultados óptimos. Puedes mirar en http://www.midconverter.com o en http://www.mp3-converter.biz a ver si encuentras la aplicación que más se ajuste a tus necesidades. | |
| Por otro lado, también puedes recurrir a sitios web donde te puedes descargar sonidos digitalizados de los más variados ámbitos: explosiones,truenos, ladridos, sonidos de oficina,etc,etc. Estas web son almacenes de recursos muy interesantes para explorar. Ejemplos son:<br>http://recursos.cnice.mec.es/bancoimagenes/sonidos<br>http://www.soundsnap.com<br>http://www.flashkit.com/soundfx<br>http://www.hispasonic.com<br>http://3dcafe.com/asp/sounds.asp<br>http://www.findsounds.com<br>http://www,therecordist.com/pages/downloads.html<br>http://sounds.wavcentral.com,<br><br>Y,si queremos midis: http://www.midisite.co.uk.<br>Si queremos archivos en formato .mod, podemos ir a http://modarchive.org ó http://www.modplug.com (donde podemos encontrar también software de creacion de ficheros mod), o http://www.mirsoft.info.<br><br>Otro recurso a tener en cuenta es la posibilidad de utilizar música libre (al igual que lo es el software), descargable de Internet de forma totalmente legal y gratuita, fomentando además la promoción de grupos y artistas que apoyan este tipo de iniciativas. Puedes echarle un ojo a http://www.musicalibre.info, http://www.musicalibre.es, http://www.jamendo.com, y si quieres más, hay un listado completo en http://parolas.thebbs.org/sincanon/Musica_libre.html | |
| ***EDITORES DE VIDEO*** | |
| **Movie Maker**<br>(de serie en Windows, pero no libre) | Lo que es el MsPaint en el mundo de los editores gráficos, es el Movie Maker en el mundo de los editores de video. Elemental, sencillo, con pocas prestaciones pero las más usuales y eficaz. Para todos aquellos que no deseen elaborar videos demasiado complejos y que se contenten con un montaje sencillo y efectivo. Ideal para comenzar. Viene instalado de serie en Windows. |
| **Kino** (http://www.kinodv.org) | Editor de vídeo no lineal libre para GNU/Linux. Sería el equivalente del Movie Maker de Windows, ya que su público es el usuario doméstico. |
| **Pitivi** (http://www.pitivi.org) | Otro editor de vídeo no lineal libre para GNU/Linux |
| **Kdenlive** (http://www.kdenlive.org) | Otro editor de vídeo no lineal libre para GNU/Linux |
| **LiVES** (http://lives.sourceforge.net) | Otro editor de vídeo no lineal libre para GNU/Linux |
| **Avidemux** | Programa libre y multiplataforma de edición de vídeo lineal, similar al |

| | |
|---|---|
| (http://www.avidemux.org) | VirtualDub de Windows. Permite modificar, cortar y aplicar filtros a vídeos codificados anteriormente. |
| **VirtualDub** (http://www.virtualdub.org) | Editor de vídeo lineal libre (pero sólo para Windows) que ofrece las mismas funcionalidades y su eficiencia y calidad son comparables a la de otros programas privativos del sector, como Adobe Premiere. Otro ejemplo de programa libre que no tiene nada que envidiar a los "monstruos" de la rama. |
| **Cinelerra** (http://www.cinelerra.org) | Otro estupendo editor de video libre, algo más complejo de utilizar (y más completo) que los anteriores. |
| **Wax** (http://www.debugmode.com/wax) | Este programa no es propiamente un editor de video sino un programa de composición de efectos especiales de video 2D y 3D. Puede utilizarse como programa independiente o como plugin de Premiere. Otro programa libre y multiplataforma de edición de video y tratamiento de efectos especiales, con soporte para OpenGL y OpenML (http://www.khronos.org/openml), es **Jahshaka** (http://www.jahshaka.org). |
| **Cinepaint** (http://www.cinepaint.org) | Más que un editor de video al uso, este programa es más bien un editor de fotogramas individuales de una película, siendo pues más similar a un editor de bitmaps realmente, como Gimp. |
| **LimSee2**. (http://limsee2.gforge.inria.fr) | Herramienta libre de autoría basado en el lenguaje SMIL (estándar abierto internacional - http://www.w3.org/AudioVideo/ - ), el cual permite integrar audio, video, imágenes, texto o cualquier otro contenido multimedia dentro de una presentación multimedia. Los navegadores web estándar son capaces de reproducir contenido SMIL, pero un reproductor dedicado a utilizar puede ser **Ambulant** (http://ambulant.sourceforge.net) |
| **Studio** (http://www.pinnaclesys.com) | Aplicación ideal para la edición de video caseros. Permite todo tipo de prestaciones para el montaje y tratamiento de las capturas y grabaciones digitales. |
| **Premiere** (http://www.adobe.com) | Competencia directa del Pinnacle Studio, aunque tal vez orientado un poco más hacia el mercado profesional: ofrece más funcionalidad que el Studio y es más utilizado para películas de larga duración y con un montaje más elaborado. |

Si todavía no tienes suficiente con la lista anterior (ni con la de Gpwiki), todavía podrás encontrar una cantidad brutal de enlaces a muchísimas más herramientas de diseño y multimedia en el espectacular post del foro de GameDevelopers:http://portalxuri.dyndns.org/gamedevelopers/modules.php?name=Forums&file=viewtopic&t=3141&highlight=mont%F3n++informaci%F3n+%FAtil

Finalmente, te comento que si quieres tener una lista bastante completa de programas libres, no ya solamente de los ámbitos tratados aquí sino de cualquier ámbito (procesadores de texto, hojas de cálculo, juegos,etc) que funcionen en Windows –y en Linux también-, échale un vistazo a la estupenda web http://www.cdlibre.org, y también a la no menos fantástica http://alts.homelinux.net.

Es más, si lo que quieres es tener un entorno donde ya tengas preinstalados la mayoría de programas libres para multimedia que he mencionado anteriormente (Inkscape, Skencil, Gimp, Blender, Kino, Cinelerra, Audacity, Rosegarden, Ardour, Hydrogen,...) sin preocuparte de descargarlos, de manera que estés listo para empezar a trabajar inmediatamente, puedes encontrar bastantes distribuciones de Linux dedicadas al mundo gráfico/video/3D/sonido con todo preparado y configurado. Incluso en formato Live-CD, para que no tengas que cambiar nada de tu ordenador. Échale un ojo por ejemplo a UbuntuStudio (http://www.ubuntustudio.org). También existe una distribución especializada solamente en software musical digna de mención: Musix (http://www.musix.org.ar). Si quieres conocer más distribuciones de este estilo, realiza una búsqueda por la base de datos de distribuciones de Distrowatch: http://www.distrowatch.com/search.php

De todas maneras, no está de más comentar (algo obvio, por otra parte) que es que en la creación y desarrollo de un videojuego medianamente complejo las tareas se han de dividir y repartir entre diversos trabajadores, ya que si no es imposible tirar adelante un proyecto profesional. Es decir, la plantilla estará formada por guionistas, los cuales habrán pensado, escrito y corregido toda la historia y sus personajes mediante técnicas al efecto -como storyboards-,; estará formada también por personal diseñador gráfico, que se dedicará a crear la ambientación (personajes, paisajes,etc) con las herramientas que se acaban de enumerar anteriormente; también puede haber parte de

los trabajadores dedicados a la creación musical; y un pequeño grupo dentro de la plantilla será el de programadores propiamente dichos, que exclusivamente se dedicarán a programar. Pueden haber más departamentos, y todos ellos serán coordinados finalmente por una comisión jefe de proyecto. Es decir, que un programador profesional no tiene por qué conocer las herramientas de diseño y viceversa.

No obstante, como nosotros desarrollaremos videojuegos amateur, donde todo lo tendremos que hacer nosotros, será necesario tocar muchas teclas y tener mínimas nociones de todo un poco: redacción de guiones, diseño gráfico, música, programación, planificación de proyectos, etc. Una ardua y completa labor, pero a la vez muy gratificante.