Froguenture

Desarrollo de Froguenture: Videojuego con generacion procedural

Nacho Herrero Espelt
Grupo B
2do DAM

Resumen del proyecto

Este proyecto se basa en el desarrollo de un videojuego roguelike con generación procedural llamado Froguenture, donde el protagonista Flubs tiene la misión de adentrarse en el bosque en busca de las semillas doradas. La generación aleatoria permite que cada partida de Froguenture sea única y diferente gracias a la gran variedad de situaciones y escenarios que se pueden generar.

El objetivo de este proyecto es crear un videojuego desde 0, implementando técnicas de generación procedural como el algoritmo Drunked Walk para generar los niveles, además de otras muchas técnicas habituales dentro del desarrollo de videojuegos.

Para conseguir este objetivo se utiliza una metodología ágil utilizando Kanban para la gestión de tareas y Github para el control de versiones

Como conclusión, este proyecto ha permitido profundizar en el desarrollo de videojuegos, concretamente en la generación procedural que es un concepto muy recurrente en los últimos años en el sector. Este proyecto además continuará en constante desarrollo con nuevas actualizaciones una vez sacada su versión inicial.

Palabras clave

- Frogunture
- Drunked Walk
- Procedural
- Aleatorio
- Items
- Roguelike

Abstract

This project is based on the development of a procedurally generated roguelike video game called Froguenture, where the protagonist, Flubs, embarks on a mission to venture into the forest in search of the Golden Seed. The random generation ensures that each playthrough of Froguenture is unique and different, thanks to the wide variety of situations and scenarios that can be generated.

The goal of this project is to create a video game from scratch, implementing procedural generation techniques such as the Drunkard's Walk algorithm to generate levels, along with many other common techniques in video game development.

To achieve this objective, an agile methodology is used, employing Kanban for task management and GitHub for version control.

In conclusion, this project has allowed for a deeper exploration of video game development, particularly in procedural generation, a concept that has become increasingly prominent in the industry in recent years. Additionally, this project will continue to evolve with new updates following its initial release.

KeyWords

- Frogunture
- Drunked Walk
- Procedural
- Random
- Items
- Roguelike

ÍNDICE

Lista de figuras

1. Introducción

1.1 Contexto

Froguenture es un videojuego del género *roguelike*, centrado en la generación procedural de niveles, en este proyecto, se generará mediante el algoritmo **Drunked Walk** y consta de un sistema de mejoras de habilidades basadas en cartas.

Esta decisión viene a partir de que en los últimos años, los *roguelikes* han ganado una enorme popularidad, posicionándose como uno de los géneros favoritos entre los jugadores. Títulos como *Hades*, *The Binding of Isaac*, *Cult of the Lamb* o *Enter the Gungeon* han dejado una fuerte huella en la industria gracias a sus mecánicas profundas, su dificultad progresiva y su alta rejugabilidad.

Uno de los pilares fundamentales de este género es la **generación procedural**, una técnica que permite crear niveles distintos de forma automática en cada nueva partida. Gracias a esto, cada recorrido es único, lo que aumenta la rejugabilidad y el interés del jugador. Además, esta metodología reduce la necesidad de diseñar niveles manualmente, permitiendo a los desarrolladores construir mundos complejos y coherentes a partir de un conjunto de reglas predefinidas.

1.2 Justificación

Este proyecto ha sido escogido debido a varios factores, el primero siendo la popularidad del género **Roguelike**. Este género se caracteriza por sus niveles generados proceduralmente y sus mecánicas que fomentan la rejugabilidad.

este género se ha hecho un gran espacio en el sector gracias a su rejugabilidad y su capacidad de ofrecer experiencias distintas en cada partida, haciendo de este sector muy demandado por la comunidad. Este proyecto busca contribuir a este catálogo de juegos ofreciendo una experiencia para los jugadores.

Esta decisión se debe además al objetivo de **desarrollar y mejorar habilidades.** Con este proyecto se busca explorar y aplicar técnicas avanzadas en el desarrollo de videojuegos, como la generación procedural, la creación de sistemas propios y el trabajo en el apartado artístico además de aplicar las ya conocidas anteriormente.

1.3 Objetivos

Los objetivos que se pretenden conseguir en este proyecto son los siguientes:

- **Objetivo general:** Con este proyecto, el principal objetivo es crear un videojuego 2.5D aplicando técnicas de generación procedural y otras técnicas de desarrollo de videojuegos.

  A nivel general los objetivos que se plantean son los siguientes:

  - Aplicar un sistema de generación procedural para la creación de los niveles
  - Sensación de juego agradable
  - Escenarios y menús estéticos
  - Gran posibilidad de opciones y rejugabilidad
  - Un flujo de juego bueno y coherente

- **Objetivos específicos:**

  Los objetivos específicos sobre el proyecto son los siguientes:

  - Usar el sistema *Drunked Walk* para la generación procedural
  - Controlar las colisiones y el entorno al sistema procedural para que no interfieran en el juego
  - Crear un sistema de enemigos inteligente con un comportamiento propio y detallado
  - Un sistema de objetos extenso y optimizado con sus respectivos efectos y propiedades
  - Gestionar los datos de guardado del jugador para que pueda continuar el juego en un futuro y mostrar sus avances

1.4 Estrategia y planificación del proyecto

Decisión estratégica:

Desde el inicio del proyecto,se optó por **desarrollar el videojuego desde cero**, sin el uso de plantillas. Esto permite un mayor control sobre la implementación de las mecánicas, especialmente en la **generación procedural de niveles y mapas**, adaptándola específicamente a las necesidades del juego.

Estudio de viabilidad:

En cuanto a la viabilidad del proyecto, es un proyecto ambicioso ya que la mayoría de los juegos actuales suele tener un tiempo estimado de 2 a 3 años, haciendo del tiempo el problema principal en el desarrollo.

A rasgos económicos, este juego podríamos decir que no se trata de un juego de un nivel comercial, y por lo tanto no sería económicamente rentable si el juego se publica a un precio gratuito, pero debido a la gran popularidad de los juegos del género se puede llegar a conocer de manera rápida y eficaz y dar paso a que los jugadores conozcan otros títulos propios anteriores o que puedan ser desarrollados en un futuro.

1.5 Metodología de trabajo

Para gestionar el desarrollo del proyecto he optado por una metodología más ágil utilizando herramientas para facilitar la organización:

Kanban con Trello: Para usar la metodología kanban he utilizado la web Trello. Este método me permite organizarme de una manera rápida y sencilla donde defino las tareas por hacer y le asignó los estados: Por hacer, En curso y Listo.

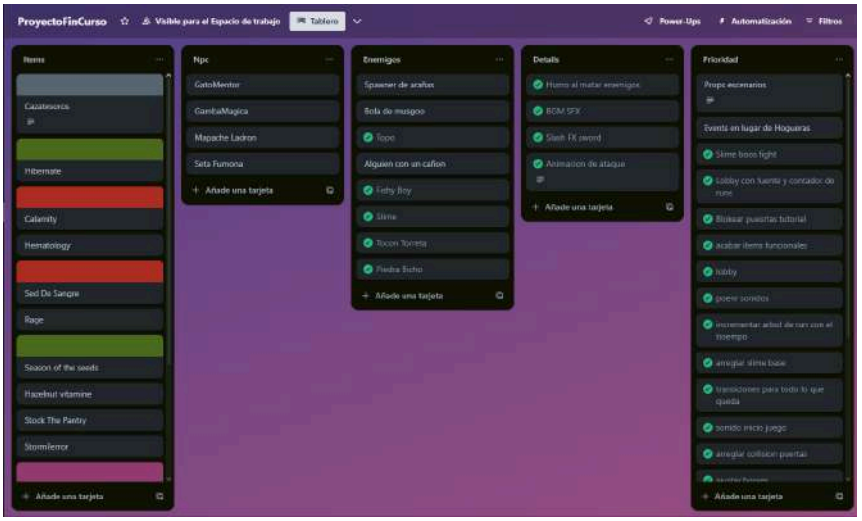

Figura 1

GitHub: Para el control de versiones estoy utilizando github como herramienta ya que me permite crear repositorios con el código del juego y tener un control de versiones con los cambios realizados

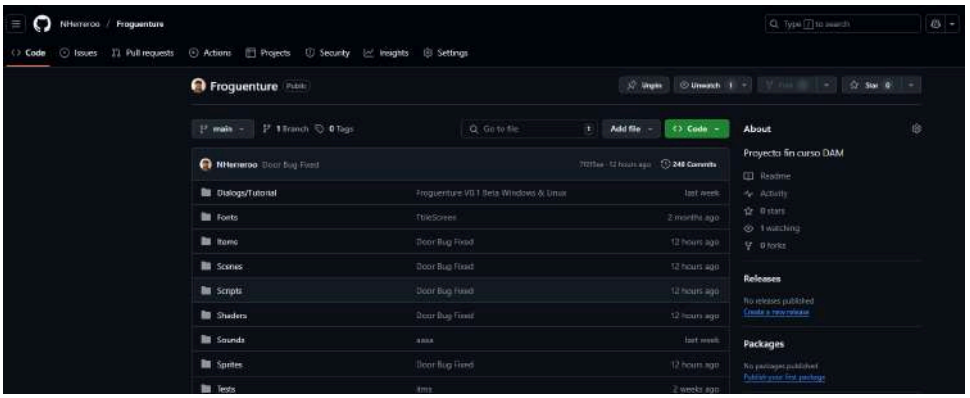

Figura 2

1.6 Estudio económico y presupuestario

Desarrollar un videojuego suele tener un coste elevado, tanto en software como en hardware, además del tiempo de trabajo. Sin embargo, este proyecto ha sido desarrollado de manera **independiente**, lo que ha permitido reducir los costes utilizando herramientas de software libre.

Aunque el desarrollo del juego no ha requerido inversiones explícitamente, sí se ha utilizado hardware y material propio que contaremos como “necesario” para desarrollar este proyecto.

El equipo principal utilizado para el desarrollo ha sido:

PC de desarrollo (CPU, RAM, GPU, SSD)	
Procesador: Intel i5	100€
RAM: 16GB DDR4	100 €
Tarjeta Gráfica: NVIDIA GTX 1650	250€
Almacenamiento SSD: 1 TB	60€
Total	510€

Periféricos	
Monitor	110€
Teclado	50 €
Raton	20€
Total	180€

Audio y Musica	
Tarjeta de sonido M Track Duo	60€
Guitarra Fender Player	900 €
Teclado MIDI AKAI	70€
Total	1030€

Diseño y arte	
Tableta Grafica Wacoom Bamboo	65€

Al tratarse de un juego independiente desarrollado por una sola persona, se han evitado costes asociados a salarios de equipo. Sin embargo, para calcular el valor económico del desarrollo, podemos estimar los costes si se hubieran contratado profesionales:

Programador/a	1.900€ (6 Meses) → 11.400€ (Aprox)
Arte y diseño gráfico	20€ → (1h) (250h) → 5.000€
Música y sonido	100€ por pista → 1.500€ (10 Pistas y sonido)
TOTAL	17900€ (Aprox)

Aunque el coste del desarrollo no es algo que se pueda saber con exactitud podemos hacer una pequeña estimación para hacernos una idea de cuál sería el presupuesto para desarrollar este juego.

PC de desarrollo (CPU, RAM, GPU, SSD)	510€
Periféricos	180€
Audio y Musica	1030€
Diseño y arte	65€
Especialistas	17900€

TOTAL	19.685€

1.7 Herramientas principales

Godot Engine: Godot es el motor de videojuegos sobre el cual se ha trabajado este proyecto, este motor es una aplicación de código abierto que se ha popularizado en el sector después del cambio de política de Unity (El motor por excelencia de videojuegos).

Godot además de ser una herramienta de código abierto muy potente, es un software extremadamente ligero (200mb) y sin necesidad de una instalación previa de gran tamaño como Unity o Unreal Engine.

Actualmente no es el motor principal para la creación de videojuegos pero la comunidad está creciendo exponencialmente y cuenta con una gran librería de Assets gratuitos, shaders, foros…

Krita: Krita es un software de código abierto con el que se han dibujado todos los personajes, items, ambientación…

Este programa de dibujo cuenta con una amplia colección de pinceles y técnicas de arte digital que nos permite realizar estas técnicas de forma ‘sencilla’ para crear la parte visual del videojuego.

FL Studio: es un programa dedicado a la producción de sonido. Este permite componer, grabar, editar, mezclar y masterizar música de forma visual. Además permite utilizar instrumentos virtuales para realizar los sonidos.

Modelio: Este software de código libre nos permite diseñar esquemas y diagramas UML que nos ayudarán a lo largo del documento a entender varios aspectos del juego.

2. Descripción del proyecto

2.1 Diseño del videojuego

Para hablar del diseño de Froguenture, primero es necesario comprender la base sobre la que se construye el proyecto. Como se ha mencionado anteriormente, el juego cuenta con un sistema de generación procedural, que es uno de los pilares fundamentales del género roguelike.

A continuación, se explica en qué consiste la generación procedural y los distintos tipos que existen, así como su aplicación dentro del diseño del juego

¿Qué es la generación procedural?

La generación procedural es una técnica usada en el desarrollo de videojuegos que utiliza algoritmos para crear datos de manera automática y aleatoria. Estos datos generados pueden ser: terrenos, niveles, mapas, patrones o cualquier elemento del juego, evitando la necesidad de diseñarlos manualmente.

La particularidad de la generación procedural se basa en la capacidad para ofrecer experiencias únicas en cada partida, ya que los jugadores se enfrentan a niveles distintos cada vez. Además, esta técnica es especialmente útil para optimizar el desarrollo, ya que permite crear una gran cantidad de posibilidades automáticamente sin tener que hacerlo manualmente.

Tipos de generación procedural

Hay muchísimos tipos de patrones y algoritmos para generar mapas procedurales. Cada uno tiene sus propiedades únicas y están enfocados de manera distinta dependiendo del tipo de generación. Las más concurrentes son:

- **Perlin Noise:** La generación Perlin Noise esta basada en un algoritmo que genera un patrón de ruido en blanco y negro donde los colores mas oscuros representan zonas altas como montañas y los colores mas claros representan zonas mas bajas como llanuras, esplanadas, valles...

  Este patrón se ha usado en juegos muy conocidos como Minecraft o No Man's Sky ya que principalmente está pensado para la generación de mundos abiertos.

- **Wave Function Collapse (*Colapso de Función de Onda*):** El algoritmo *Wave Function Collapse* (WFC) es un método basado en la propagación de restricciones y en la superposición de patrones. Funciona analizando un conjunto de *tiles* (o patrones) predeterminados y generando un mapa que respeta las reglas de conexión entre ellos

  Este método es particularmente útil para la generación de mapas coherentes y estructurados, como mazmorras o niveles en juegos de plataformas, ya que permite garantizar que las partes generadas tengan sentido y se conecten adecuadamente. Juegos que utilizan esta técnica incluyen *Caves of Qud* y *Bad North*, donde los mapas deben mantener una estructura lógica.

- **Drunk Walk:** El famoso algoritmo Drunk Walk es uno de los más conocidos, si no el que más, y además de ser el más sencillo es el método utilizado en este proyecto.

  El algoritmo simula el recorrido de una "**persona borracha**" (*Drunk Person*) dentro de una matriz. La simulación comienza con esta persona en el centro de la matriz, y desde allí se mueve en una de cuatro direcciones posibles: arriba, abajo, izquierda o derecha.

  Cada paso que da esta persona se decide de manera aleatoria hasta que se finalice el número de pasos.

  A medida que el esta persona va avanzando, marca el terreno(matriz) dando a entender que ha pasado por esa casilla, dando como resultado una matriz que marca las posiciones por las que ha pasado la persona.

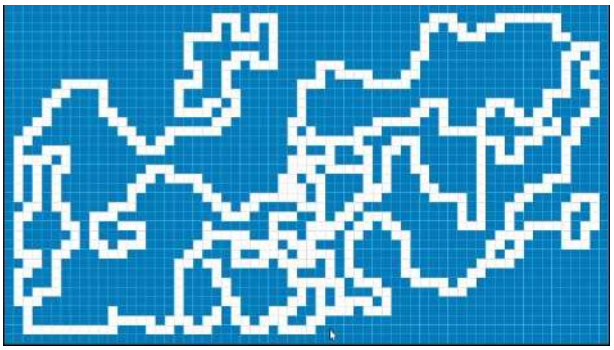

Figura 3

2.1.1 Arquitectura del proyecto

Froguenture está estructurado de una manera sencilla y eficiente basada en una arquitectura de **cliente**, es decir, todas las interacciones con el juego y datos de guardado se gestionan de forma local en su ordenador evitando depender de servidores externos.

Para el desarrollo de este proyecto se han utilizado varios **patrones de diseño**, siendo los dos más relevantes el **patrón Singleton** y el **patrón Template**.

El **patrón Singleton** es uno de los más comunes en el desarrollo de videojuegos, y en este caso se ha implementado en muchos puntos del proyecto. Como Godot organiza los elementos en forma de árbol jerárquico basada en nodos y escenas, puede resultar complejo compartir o acceder a cierta información entre distintas partes del código. Gracias al uso del Singleton, es posible **almacenar datos globales** (como estadísticas del jugador, estados del juego o configuraciones generales) y acceder a ellos desde cualquier parte del proyecto.

Clases Singleton utilizadas:

- **Global**: Guarda datos globales de la partida
- **Player**: Guarda las estadísticas generales del jugador
- **Apply items**: Se utiliza para aplicar efectos sobre los ítems y se maneja junto al singleton del Player
- **Config**: Guarda la configuración del juego
- **Events**: Define señales que se utiliza puntualmente
- **Spawn Points**: Contiene los patrones de generación de enemigos dependiendo de la habitación
- **Save System**: Maneja el guardado, cargado y borrado tanto para la partida como para la configuración

En cuanto al **patrón Template**, consiste a grandes rasgos en crear un objeto base, en este caso, una escena, que comparte un comportamiento común. Esta escena actúa como plantilla, y a partir de ella se **duplica y adapta** para añadir comportamientos específicos.

En el proyecto se aplica, por ejemplo, en los **ítems** del juego: todos comparten una misma estructura pero cada uno tiene un efecto distinto al recogerse. Gracias a este patrón, se reutiliza la misma base y solo se modifica el comportamiento único de cada objeto.

2.1.2 Concepto y narrativa

El videojuego desarrollado trae como título ***Froguenture*****,** de la fusión de **Frog** + **Rogue (Roguelike)** + **Adventure.** Este título trae como característica principal la generación de niveles procedurales y los juegos de cartas **TCG** como ***Magic The Gathering*****.**

Froguenture cuenta con una infinidad de posibilidades a la hora de jugar gracias a la variedad de objetos y situaciones aleatorias que pueden llegar a suceder.

En **Froguenture**, el jugador asume el papel de **Flubs**, una valiente rana que habita en el bosque. Con el debilitamiento del Gran Árbol, la vida en el bosque ha comenzado a desvanecerse, dando como fruto un paisaje devastado por el vacío, llegando a alterar a otros animales del bosque.

La única esperanza de restaurar el equilibrio son las **Semillas Doradas**, reliquias que, en tiempos antiguos, fueron esparcidas por el viento desde lo más alto del Gran Árbol.

Ahora, Flubs deberá embarcarse en una peligrosa aventura para encontrarlas y devolver la vitalidad a su hogar antes de que sea demasiado tarde topándose con diversos personajes por el camino que le ofrecerán ayuda para su aventura.

2.1.3 Diseño de personajes, entorno e items

Para el diseño de los **personajes** y del **entorno** se ha tomado como referencia juegos como *Cult of the Lamb* para la **estética y caracterización visual** de los personajes y escenarios.

Por otro lado, para el sistema de **generación del mapa**, tomé además como referencia *Inscryption*, en concreto su estructura de mapa en forma de **árbol de decisiones**, que permite representar de manera visual y clara las diferentes rutas que el jugador puede tomar. Estas influencias han ayudado a definir la identidad visual y estructural del juego.

Diseño de personajes

Personaje Jugable (Flubs):

El protagonista del juego es **Flubs**, una rana con una apariencia simpática con vestimenta medieval. Su diseño está inspirado en la carta de ***Magic: The Gathering*** "**Flubs, the Fool**", la cual representa la carta del tarot ***"The Fool"*** (El Loco). Esta elección no es casual, al igual que el loco, Flubs simboliza el inicio de una aventura. Su estilo visual busca transmitir un aspecto simpático.

Figura 4

Personaje de Tienda (Winter):

Winter es el personaje encargado de gestionar la tienda del juego, así como las habitaciones de tesoro. Se trata de un gato con poderes psíquicos que levita sobre una manta. Cuenta con una personalidad bastante cómica y graciosa que da cierto contraste al juego.

Figura 5

Personaje del tutorial (Sakgu):

Sakgu es el personaje que encontramos durante el tutorial. Esta **tortuga chamán** actúa como nuestra guía en los inicios del juego. Tiene un carácter misterioso y transmite una actitud de sabiduría y calma.

Figura 6

Búho Místico (Boss):

El búho místico corresponde a uno de los jefes finales del juego, y cuenta con una presencia imponente, controla sus ataques con su mente lo cual puede llevar a cabo que se debilite con facilidad.

Figura 7

Rey Slime (Boss):

El Rey Slime es otro de los enemigos finales del juego y está principalmente basado en el primer jefe del conocido juego *Terraria.* Este jefe cuenta con un aspecto algo cómico debido a sus expresiones faciales

Figura 8

Diseño del entorno

El entorno del videojuego se desarrolla en un **bosque con elementos de estética medieval**, esto se puede ver en estructuras como muros de piedra, fuentes y otros detalles del juego. Esta ambientación se inspira en la atmósfera visual de *Cult of the Lamb*, pero adaptándola a un enfoque más **medieval**.

Figura 9

Diseño de los ítems

Para el diseño de los ítems, se han tenido en cuenta dos grandes inspiraciones. La primera es *Magic: The Gathering*, que ha servido como referencia para la estructura general de las cartas y la segunda inspiración son las **cartas del tarot**, utilizadas como cartas con diferente arte pero manteniendo el mismo funcionamiento. Este recurso es muy común en juegos independientes como *Balatro* o *The Binding of Isaac* o series como *Jojo's Bizarre Adventure* donde las cartas del tarot juegan un papel importante.

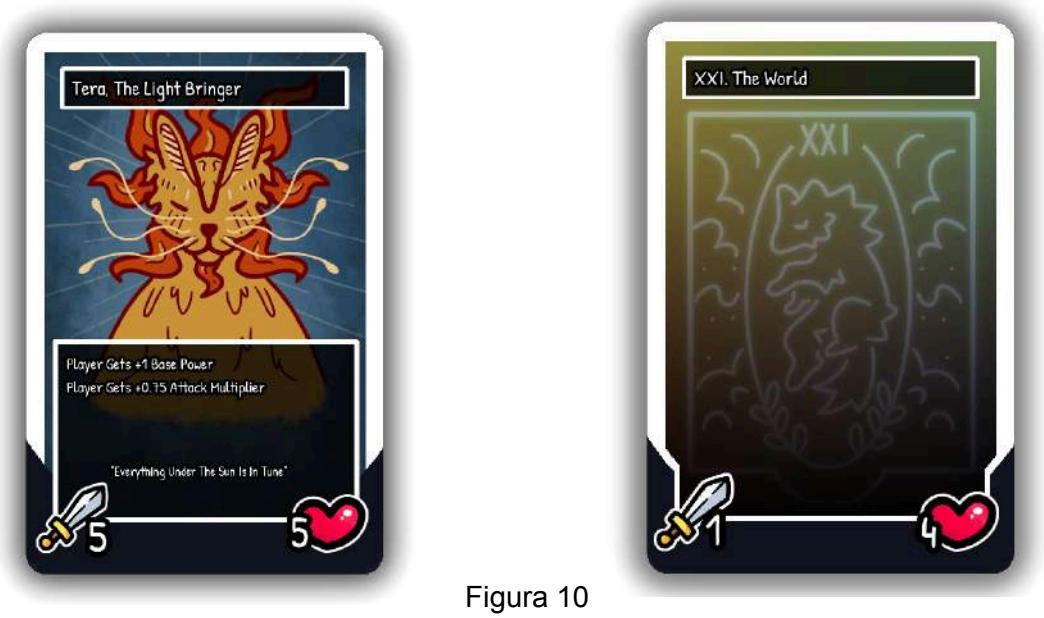

Figura 10

2.1.4 Mecánicas de juego

- **Generación Procedural:** El juego cuenta con una generación procedural para los niveles basada en al algoritmo Drunked Walk, está algoritmo nos permite generar un nivel aleatorio a partir de una matriz. En esta generación está modificada y más desarrollada para proporcionar mapas más carismáticos como habitaciones de pelea, tienda, tesoro o boss.

- **mapa procedural:** Al iniciar una partida o acabar un nivel completo, se abrirá un mapa completamente aleatorio que se generará exclusivamente al principio de la Run. Este mapa usa otro método de generación procedural basado en **DAG(Directed Acyclic Graph)** ya que cuenta con una estructura basada en nodos además de ser jerárquica. Este mapa generará una serie de niveles conectados entre sí, permitiéndonos varias opciones a la hora de escoger nuestro camino.

  El jugador no podrá volver hacia atrás, y solo podrá seleccionar los niveles adyacentes al que está:

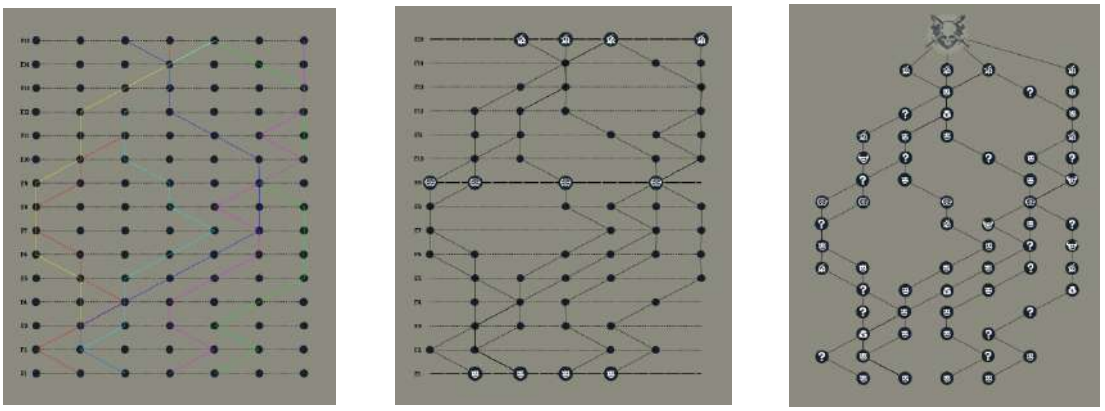

Figura 11

- **Sistema de items:** Como todo juego Roguelike, debe haber un sistema de items extenso para manejar tanto la escalabilidad del personaje, como las sinergias que hacen estos items/objetos a lo largo de la partida permitiendo combinar varios efectos para hacer combinaciones extrañas y potentes.

  Este sistema esta basado en el juego de cartas *Magic the gathering* y cuenta con una gran variedad de cartas de varios colores (Blanco, Negro, Azul, Rojo, Verde, Incolora). Cada una de estos colores representa un tipo de juego.

  - Negro: Daño venenoso
  - Blanco: Velocidad de ataque
  - Azul: Escudos de protección
  - Rojo: Impacto crítico
  - Verde: Recuperación de vida
  - Incoloro: Habilidades genéricas

Además de los colores, existen diferentes tipos de carta (Criatura, Artefacto, Hechizo, Instantáneo…). Estas cartas de por si no tienen ningún efecto adicional, pero pueden haber cartas que escalen en base a estas (*+5 de año por cada carta de Artefacto*)

- **Sistema de combate:** Los combates son algo esencial en los juegos de acción, en este caso los combates están compuestos por varios factores: La sala, el personaje/jugador y los enemigos.

  El personaje principal cuenta con un ataque Melee donde utiliza su espada para derrotar a los enemigos. El personaje, a pesar de tener un ataque básico, cuenta con un ataque especial al realizar su tercer ataque consecutivo a modo de combo. Además este personaje puede aumentar sus estadísticas gracias a los objetos e ítems a lo largo de la partida para enriquecer este sistema de batalla.

2.1.5 Escenarios y niveles

Este videojuego cuenta con una gran variedad de escenarios, como se ha mencionado anteriormente, el juego genera los niveles de forma automática y utiliza salas ya diseñadas para generar el nivel completo.

Las salas principales que se pueden generar durante la partida son de estos tipos:

- **Sala inicial**: Es la habitación inicial de la partida, el personaje aparece en esta al entrar al nivel

- **Sala de tesoro:** Al acceder esta sala nos encontraremos a un NPC el cual nos dará un sobre aleatorio en el que contiene dos ítems a elegir

- **Tienda**: Esta sala cuenta con un NPC que nos ofrecerá objetos a cambio de monedas

- **Sala de enemigo**: La sala de enemigos es la sala más habitual del nivel, esta sala será una de la colección de salas predefinidas que escoge el juego aleatoriamente para generar el nivel. Estas salas cuentan con enemigos y obstáculos además de sus puertas adyacentes a la ostras salas, estas puertas no se abrirás hasta que se derroten a todos los enemigos o se cumplan los objetivos

- **Sala de jefe**: En esta habitación podemos encontrar un enemigo final que nos dará una recompensa mayor al resto. Esta sala es la habitación final del nivel, al finalizar, pasaremos al siguiente nivel, permitiendo avanzar por el mapa.

- **Sala normal**: Estas salas son las más comunes ya que representan el gran porcentaje de salas jugables en el juego. Hay nueve tipos de habitaciones normales y cada una cuenta con diferentes obstáculos

2.1.6 Interfaz de usuario (UI)

HUD (Head-Up-Display): es la interfaz principal del jugador, está compuesta por varios elementos que muestran la información más relevante sobre la partida y el jugador.

El HUD está compuesto por los siguientes componentes:

- **Vida y escudo**: Muestra la vida y el escudo a tiempo real del jugador, esta puede variar a lo largo de la partida dependiendo si el jugador recibe daño, o restaura la vida.

- **Estadísticas:** en la parte lateral izquierda se muestran las estadísticas del jugador en tiempo real para que el jugador tenga una visión general y pueda tener un mejor control a la hora de seleccionar objetos.

- **Dinero:** Muestra la cantidad total de dinero obtenido, este se puede gastar en las tiendas para comprar objetos que pueden ayudar al jugador en el transcurso de la partida.

- **Mini Mapa**: Muestra el mapa del nivel, solamente se muestra las habitaciones disponibles a una posición de distancia desde la posición del jugador.

- **Efectos de daño:** Cuando el jugador recibe daño por parte de algún enemigo, se muestra un degradado rojo para indicar que la vida ha disminuido. Aunque no se muestra constantemente en el Hud, es una parte fundamental de este.

Figura 12

2.1.7 Diseño de sonido y Música

En cuanto al diseño de sonido, todas las pistas musicales del juego son composiciones originales creadas para el proyecto. La mayoría de los efectos de sonido también han sido producidos de forma propia, aunque algunos han sido descargados a través de la plataforma **Freesound.org**, bajo licencias **Creative Commons 0 (CC0)** y **Creative Commons Attribution 3.0 (CC BY 3.0)**.

La licencia **CC0** permite usar los sonidos libremente sin necesidad de acreditar al autor, mientras que la **CC BY 3.0** permite su uso y distribución siempre que se mencione al autor original.

Composición musical:

Para la creación de la música del juego se ha utilizado **FL Studio**, un software muy conocido para la producción de música. Como el juego está ambientado en un bosque y la música se inspira en un entorno medieval, a pesar de no tener físicamente ciertos instrumentos, estos fueron tocados virtualmente. Para ello, se ha recurrido a una herramienta del propio programa llamada **Soundfonts**, que permite cargar bibliotecas de instrumentos virtuales.

Estos instrumentos pueden tocarse mediante señales **MIDI** (Musical Instrument Digital Interface), este sistema no graba el sonido como tal, sino las instrucciones, por ejemplo: qué nota se toca, cuándo, con qué intensidad... Gracias a esto, se ha podido componer cada pista musical utilizando instrumentos virtuales.

La herramienta soundfonts se ha utilizado en una infinidad de música para videojuegos, siendo **Undertale** el más conocido usando la librería de sonidos de la saga M**other y EarthBound**

Para la composición de la música de tienda no se ha utilizado instrumentos digitales, en este caso, la guitarra ha sido grabada físicamente mediante una tarjeta de sonido, a esta pista se le corrigieron los altos, medios y graves y se agregó un efecto de guitarra clásica al audio original.

Composición de efectos:

La mayoría de los efectos de sonido utilizados en el juego han sido descargados de **Freesounds.org**, como se ha mencionado anteriormente, aunque también se han creado algunos de forma manual.

Un ejemplo de esto es el **efecto de sonido al abrir un sobre**, que fue grabado arrugando un paquete de galletas Oreo frente al micrófono. Otro efecto propio, aunque no se utilizó en la versión final del juego, fue un **sonido para el botón de confirmar**, el cual consistía en golpear la anilla de una lata y modificar el audio para hacerlo más grave.

2.2 Desarrollo del videojuego

2.2.1 Configuración del proyecto

La configuración inicial del proyecto se presenta principalmente en las propiedades del proyecto dentro del propio motor.

Inicialmente el juego está pensado para jugar en pantalla completa aunque para una versión futura al usuario se le da la opción de cambiar a modo ventana dentro del menú de opciones, si el jugador lo prefiere también se puede escalar la pantalla estando en modo ventana para una mejor accesibilidad.

Para controlar al personaje y otras acciones del juego, Godot usa un sistema llamado mapa de entrada (Input Map). Este sistema permite asignar **varias teclas o botones** a una misma acción, como por ejemplo moverse, o hacer el dash. Gracias a esto, el juego puede ser compatible con **teclado, ratón o mando**, sin tener que escribir código distinto para cada uno.

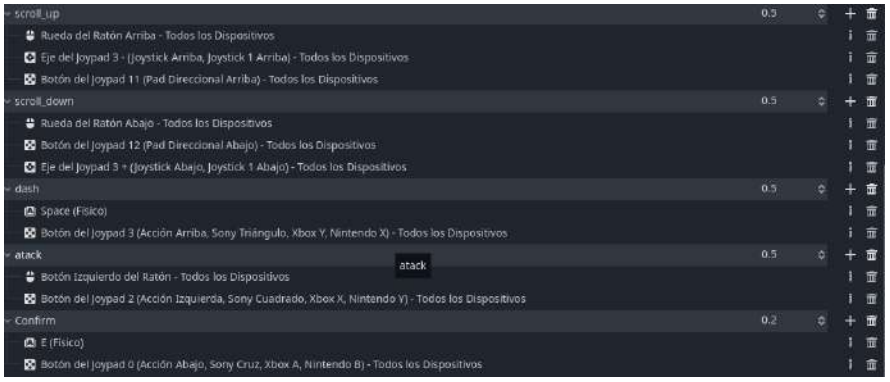

Figura 13

En la configuración del proyecto, también debemos definir los scripts de tipo **singleton**. Como se ha mencionado anteriormente, estos scripts representan una única instancia que puede ser accedida desde cualquier parte del código. Para ello, debemos asignarles un nombre en Godot, que será el que utilizaremos para referirnos a ese archivo en los scripts. Esta configuración se realiza a través de la opción **Autoload** en Godot, que nos permite cargar el script automáticamente.

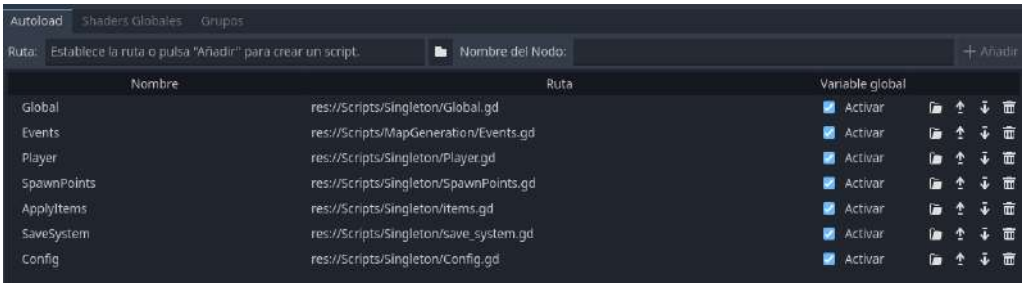

Figura 14

2.2.2 Desarrollo del videojuego

Funcionamiento del motor

El motor Godot funciona a través de un sistema basado en ***escenas*** y ***nodos***. Una escena es un conjunto de nodos organizados de manera jerárquica en forma de árbol para formar un objeto o una parte del juego.

Cada nodo dentro de una escena tiene una función específica: algunos pueden representar elementos visuales, otros manejar la lógica del juego, colisiones, sonido o interfaces. Lo interesante de este sistema es que una escena puede estar compuesta por otras escenas, lo que permite una estructura sencilla y reutilizable.

Pongamos de ejemplo una escena llamada “Jugador”, esta escena está compuesta por nodos propios del motor en forma de árbol, como por ejemplo, las colisiones, la cámara, el dibujo del personaje…

Este jugador, digamos que tiene un comportamiento que maneja los ataques, la salud y el movimiento. Para gestionar este comportamiento hay que programarlo y añadirlo a un nodo de la escena mediante un **Script**.

Los **Script** son los archivos que manejan toda la lógica y el funcionamiento del juego, estos usan su propio lenguaje (**GdScript**) y utilizan la extension **.gd**. También se puede utilizar C# pero su implementación todavía se está mejorando.

Ahora que tenemos la escena del jugador, añadiremos este a otra escena que llamaremos “Nivel” que además de la escena del jugador, tendrá los nodos que componen esta, como por ejemplo, el sonido, las monedas, el entorno…

Este ejemplo se vería así dentro del motor:

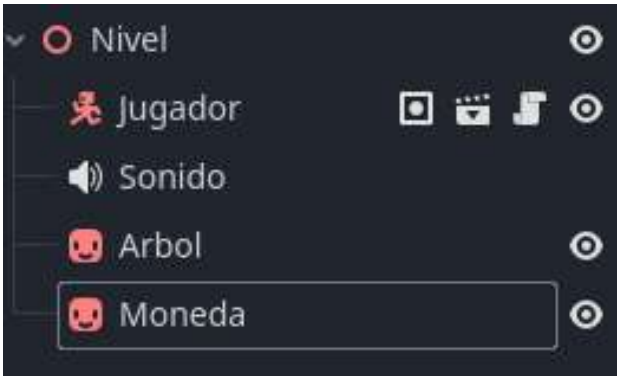

Figura 15

Flujo de juego

Antes de iniciar el flujo principal del juego, se presenta un menú principal. Si el usuario no cuenta con una partida guardada, deberá comenzar una nueva. Al crear una partida nueva, se accede primero a un tutorial (no procedural), diseñado para enseñar de forma rápida y clara las mecánicas más interesantes del juego. Una vez completado el tutorial, el jugador entra al flujo normal del juego, de la misma forma que lo haría si ya hubiera omitido el tutorial.
Dentro del proyecto, hay una escena que maneja el flujo del juego, esta escena denominada com **Run**, es la encargada de controlar el flujo del juego. Esta escena actúa como la raíz de la jerarquía de escenas, lo que significa que cualquier elemento que se dibuje en él estará por encima de todo lo demás. A partir de Run, se generan y gestionan las distintas escenas, organizándose como un árbol jerárquico.

Su función principal es gestionar la transición entre el mapa, y los niveles jugables. Cuando la escena Run se inicia, es decir se comienza una nueva partida, se genera un nuevo mapa y se desbloquea la primera planta. Además, se inician las señales con eventos clave, como la salida de una habitación o el inicio de una transición, para que el juego pueda reaccionar a las acciones del jugador.

Las **señales** son una función fundamental dentro de godot ya que como su nombre indica, nos permite enviar una señal para que el programa interprete que debe de hacer algo cuando se realiza una acción. Un ejemplo sencillo para explicar el funcionamiento de las señales, sería con los botones. Cuando el usuario pulsa un nodo de tipo **Botón**, este envía una señal indicando que ha sido pulsado y por lo tanto realizará la función específica que se le indique.

Cuando el jugador selecciona un nivel en el mapa, se emite una señal que activa el cambio de escena hacia el nivel correspondiente, dependiendo del tipo seleccionado y se oculta el mapa. De forma inversa, al completar un nivel, se emite otra señal que hace que el mapa se vuelva a mostrar y se desbloquee la siguiente habitación disponible.

Además, al ser el nodo de mayor prioridad en la jerarquía, Run también maneja las transiciones entre escenas. Para ello, cuenta con un sistema de animaciones que, al recibir la señal de transición, se crea una escena que maneja esta animación, asegurando un cambio más agradable entre las diferentes partes del juego.

Una vez completado el mapa y derrotado el jefe final, la partida se da por finalizada. En ese momento, se muestran al jugador las estadísticas resumidas de la partida. Tras ello, el sistema de autoguardado registra automáticamente estas estadísticas.

Finalizado este proceso, el jugador es redirigido a una escena llamada **Lobby**, donde puede acceder a un portal para iniciar una nueva partida, lo que lo llevará de nuevo a la escena **Run**, reiniciando así el flujo de juego.

Para visualizar el diagrama de flujo, véase el **anexo**.

Generación de niveles procedurales

La generación de niveles, como se ha mencionado anteriormente se ha utilizado el algoritmo **Drunked Walk**. Para implementar este algoritmo en este proyecto, he creado un script donde se generará toda la lógica de la generación.

Para empezar se definen las constantes de las habitaciones como caracteres, cada tipo de habitación tendrá un carácter especial para que el código interprete ese carácter como una habitación específica a la hora de dibujar el nivel por pantalla.

Las habitaciones que no son específicas como la tienda, sala inicial, tesoro… Serán habitaciones normales de batallas, estas en lugar de tener un carácter especial como podría ser **$** para las tiendas, se le asignará un **número aleatorio**, este número hará referencia a la habitación prediseñada que tenga ese número.

Una vez definido esto, crearemos el mapa por el que pasará nuestro ***hombre borracho***. Primero se crea una matriz del tamaño establecido y diremos que posicione al hombre en el centro de esta matriz. A este sujeto antes de empezar a caminar le definiremos el número de pasos que dará en la matriz, estos pasos son el número exacto de habitaciones que generará, este número de pasos puede ir siendo modificado a lo largo de la partida para generar niveles más largos y aumentar la dificultad.

Antes de que se ponga a caminar, debemos definir una serie de **normas** para generar salas especiales. Para las habitaciones de tienda y tesoro les asignaremos un número aleatorio entre los pasos que da, este número indica a qué paso se generará esa sala, es decir, si la sala de tesoros tiene asignado el número 6, cuando el hombre borracho lleve 6 pasos pintará en el mapa una tienda, es decir, la constante definida anteriormente.

Otra norma sería la **sala inicial y la del boss** que serán el primer y último paso que se hagan, al igual que las otras normas se pintara en la matriz el carácter que le pertoque.

Seguidamente de definir estas normas el hombre ya puede empezar a caminar. Desde el centro de la matriz, por cada paso definido, escoge una dirección (Arriba, Abajo, Izquierda, Derecha) y se mueve hacia la dirección escogida, una vez se ha movido, dibuja en la matriz el carácter que toque. Así se vería una representación gráfica del mapa:

Figura 16

Con todo esto ya se generaría la lógica del mapa, ahora solo queda preparar las salas que se tiene que generar a partir de este mapa además de sus puertas adyacentes a las otras habitaciones.

Para instanciar las **salas del nivel**, no las vamos a generar todas a la vez ya que consumiría una gran cantidad de recursos. En lugar de eso, pre-cargaremos en el script las rutas de todas las salas que se pueden generar y así indicarle la **ruta** de estas habitaciónes para poder instanciar las cuando queramos.

Para determinar la sala que toca instanciar se utiliza una especie de "**puntero**" que está comprobando la posición del personaje en la matriz, es decir, la sala en la que el jugador estaría jugando, una vez este puntero sabe dónde está el jugador en la matriz, consulta su celda, e instancia la habitación al momento.

El flujo de juego **no va cambiando de escenas** en ningún momento, es decir, todo el nivel es **una misma escena**, esta escena es la que se va a encargar de ir instanciando la sala en la que está el jugador dando la impresión de estar cambiando de escena, esto evita excesivos tiempos de carga y un mejor manejo a nivel de código aunque más complejo.

IMAGEN EXPLICATIVA

Cuando el personaje pasa por una puerta está accediendo a otra habitación del mismo nivel, estas puertas se generan de forma automática, no están implementadas dentro de las habitaciones ya que son escenas independientes.

Estas puertas se generan gracias al puntero mencionado anteriormente, cuando el puntero está comprobando la habitación en la que está el jugador, comprueba si tiene habitaciones adyacentes, si tiene habitaciones, generará las respectivas puertas en la habitación instanciada.

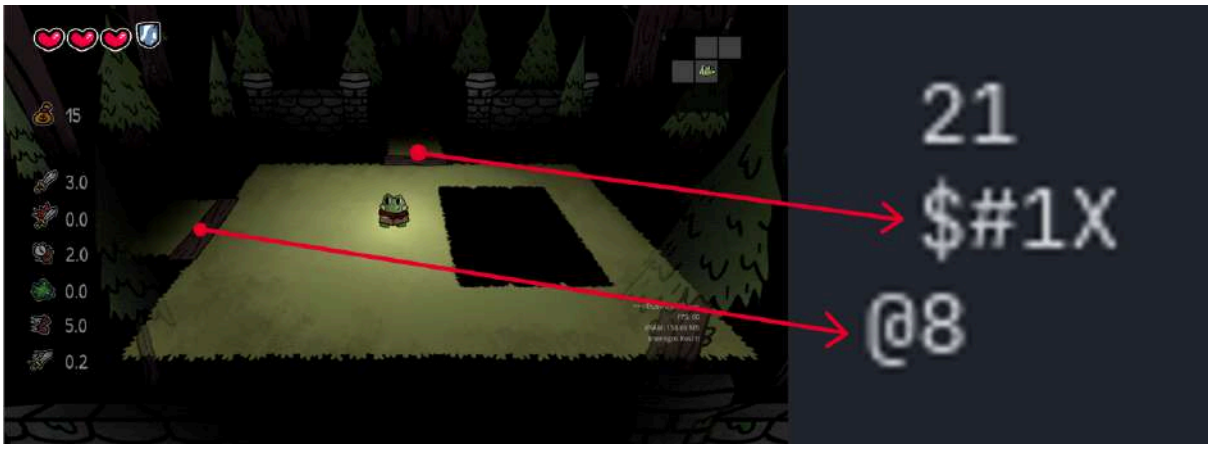

Figura 17

Finalmente cuando el personaje avanza por una puerta el puntero también sigue al jugador para que pueda ir analizando lo que necesite generar. Como el jugador cambia de habitación y tiene que mostrar la nueva, primero hay que **borrar las puertas** y la **habitación** previa a la que el jugador ha estado, para ello creamos un método encargado de actualizar el nivel y cuando detecte que se pasa por una puerta borre absolutamente todo el nivel anterior para generar el actual.

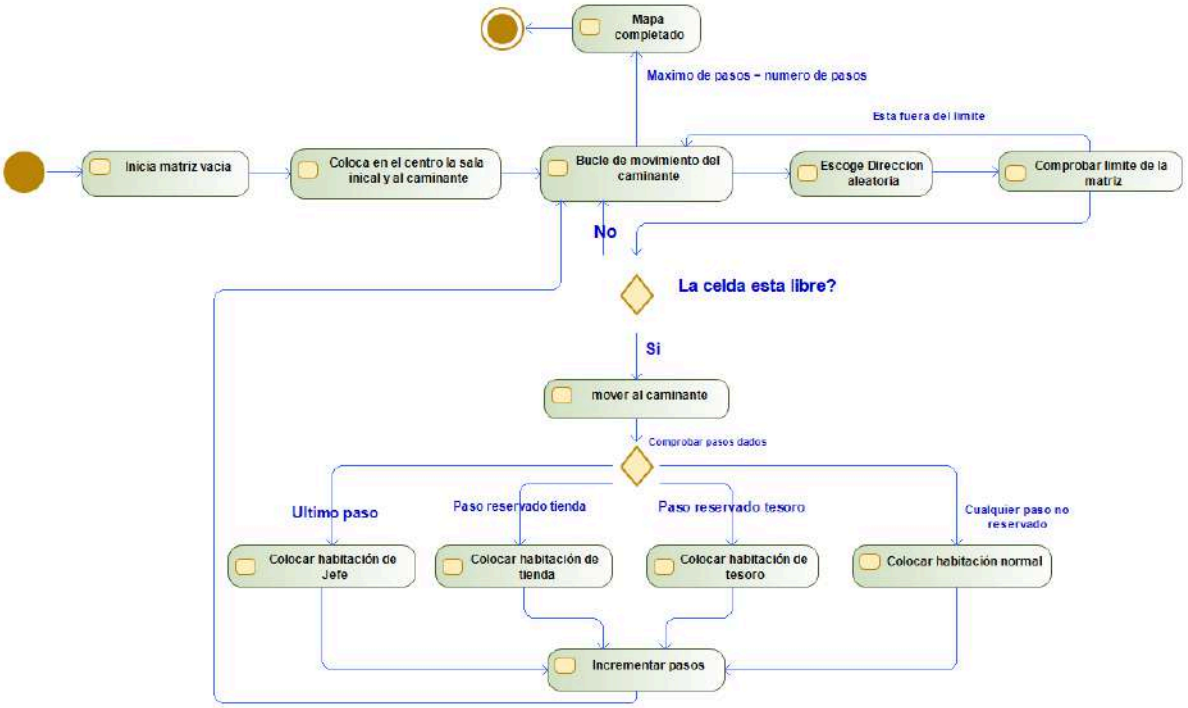

Figura 18

Mapa vertical procedural

El mapa procedural es algo básico en cualquier juego del estilo, cada videojuego lo hace de una manera en especial que permite dar un toque más especial a los juegos. En este proyecto he optado por generar un mapa de manera procedural usando un algoritmo en forma de árbol llamado **DAG(Directed Acyclic Graph).**

Este tipo de mapa está inspirado en otros títulos como **Slay The Spire** o **Incryption.** Este tipo de generación de mapa es bastante original ya que a partir de un punto inicial se crean una serie de puntos en forma de árbol donde no se pueden cruzar estos caminos y además desembocan todos en un jefe final indicando el final de la partida. Además el jugador puede elegir el camino por el que quiere avanzar impidiendo volver hacia atrás y seleccionando niveles específicos como tiendas, enemigos, tesoros…

Figura 19

Para crear este tipo de mapa en el proyecto se han utilizado 4 Scripts diferentes que forman la base de la generación del mapa, a continuación se explica detalladamente el funcionamiento de estos:

Room está dedicado a la representación lógica de una habitación. Este script es el más sencillo de estos 4 y no tiene una representación visualmente directa ya que solo maneja lógica para para almacenar datos que se usarán más adelante.

La clase Room extiende de **Resource**, un resource en godot es un objeto que nos permite usar datos reutilizables y crear menús visuales dentro del motor. Sobre todo está pensado para utilizar datos propios del motor como pueden ser las texturas, animación, shaders, nodos…

En este caso, cada sala del juego puede tener un tipo distinto, como monstruo, tesoro, hoguera, tienda o jefe final. También guarda información como su lugar en el mapa (fila, columna y posición exacta), si está conectada con otras salas, y si ha sido seleccionada por el jugador. Todo esto permite que el juego sepa cómo es cada sala y cómo debe comportarse.

MapRoom es la representación visual de las habitaciones en el mapa, es decir, los niveles que aparecen en el mapa. Este script extiende del nodo Area2D para que sea interactivo y transforma los datos de ***Room*** para mostrarlos por pantalla.

Este código se encarga principalmente de mostrar cada sala del mapa con una imagen distinta, según su tipo. También dibuja las líneas que conectan las diferentes salas entre sí, creando una red visual del camino que se puede seguir. Además, hay animaciones que ayudan a destacar las salas que ya han sido visitadas o las posibles opciones a elegir.

El sistema también permite que el jugador interactúe con el mapa. Cuando el jugador elige una sala mediante el click del ratón, esta se marca como visitada para que no se pueda volver a seleccionar.

MapGenerator es la parte más importante en la generación del mapa. Aquí es donde se crean todas las habitaciones y conexiones entre ellas.

Primero se definen algunas reglas básicas, como cuántas salas debe haber como mínimo en vertical, el ancho del mapa, cuántos caminos diferentes se van a crear, y con qué frecuencia aparece cada tipo de sala…

Una vez hecho esto, se genera una matriz donde cada espacio puede ser una posible sala. Sin embargo, no todos estos espacios se usan. Algunos se dejan vacíos para evitar que los caminos se crucen y para que el recorrido por el mapa sea más interesante, con bifurcaciones y distintas rutas posibles.

Las salas no se colocan al azar del todo: siguen ciertas normas que aseguran que los caminos siempre terminan llegando hasta el jefe final a partir de los datos que le asignamos anteriormente.

Después de colocar las salas, se crean las conexiones entre ellas, siguiendo también unas reglas específicas para que todo esté bien conectado y tenga sentido dentro del juego. Las normas que se siguen para realizar las conexiones son las siguientes:

- Cada sala debe tener al menos una conexión con otra habitación
- Los caminos nunca deben cruzarse, lo que significa que las conexiones se trazan de forma lógica y ordenada.
- Se crean bifurcaciones para dar opciones al jugador, pero siempre garantizando que exista un camino principal hacia la sala del jefe final.

Una vez que el mapa tiene su estructura, se asignan los tipos a las habitaciones usando reglas específicas. Esto se hace para mantener una estructura de juego. Por ejemplo:

- **ENEMIGO:** son comunes, pero no pueden estar todas juntas.
- **HOGUERA:** son menos frecuentes y suelen colocarse estratégicamente para ofrecer una pausa al jugador.
- **JEFE** y **TREASURE**: tienen ubicaciones fijas o casi fijas: el jefe siempre está en el último nivel, y el tesoro suele estar en una bifurcación para recompensar a los jugadores que exploran y al final del mapa antes de la batalla final.

Aquí se aplican restricciones, como evitar que dos HOGUERA estén demasiado cerca o que aparezca una sala ENEMIGO justo al lado del punto de inicio.

Map es el encargado de transformar la lógica generada por MapGenerator en elementos interactivos y visibles para el jugador. Este script trabaja en conjunto con MapRoom para mostrar el mapa.

Entre los elementos más importantes están los datos que se usan para crear cada sala del mapa, y la velocidad a la que el jugador puede moverse hacia arriba o abajo por el mapa con la cámara.

Las funciones principales de este sistema son:

- **Crear el mapa**: Se encarga de generar la estructura del mapa utilizando una lógica ya definida, y luego transforma esa información en salas reales que se colocan en pantalla.

- **Colocar las salas**: Una a una, se van añadiendo las salas al mapa para que el jugador pueda verlas e interactuar con ellas.

- **Conectar las salas**: Se dibujan líneas entre las salas para mostrar qué caminos se pueden seguir.

- **Desbloquear nuevas salas**: A medida que el jugador avanza y supera niveles, se activan nuevas partes del mapa para que pueda seguir explorando.

Además, este script maneja las interacciones del jugador con el mapa, detectando cuando selecciona una habitación y emitiendo señales para notificar al juego principal que debe realizar una acción asociada a esa habitación (por ejemplo, iniciar un nivel).

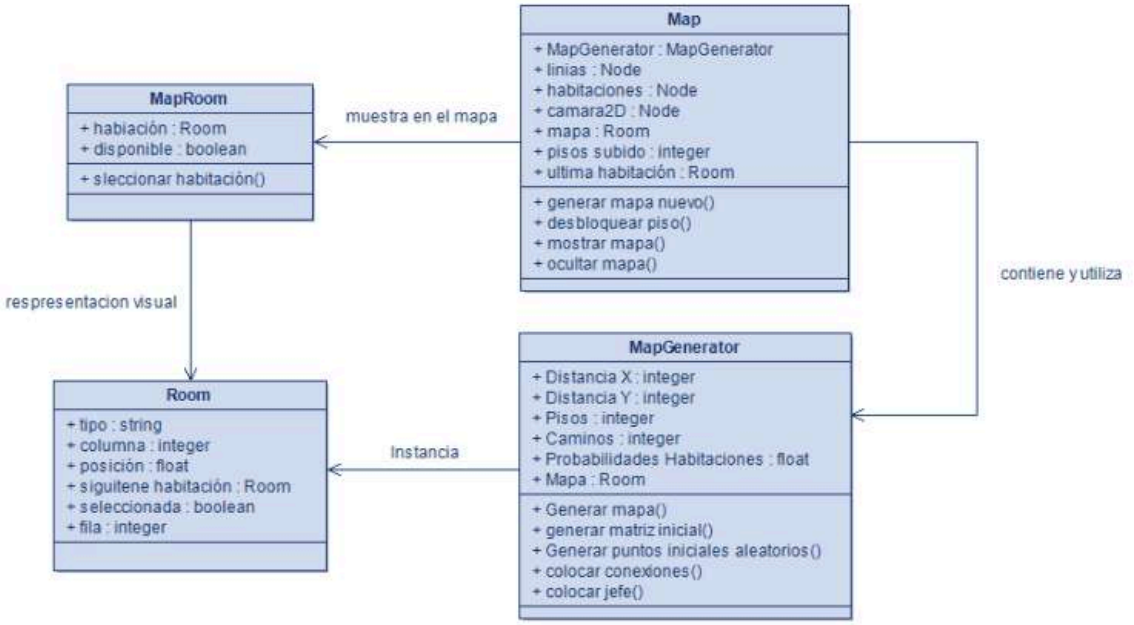

Figura 20

Programación del jugador

El script **Player.gd** es el encargado de todo lo que el jugador puede hacer: moverse, atacar, esquivar y gestionar su salud. Es por eso que es uno de los códigos más importantes en el videojuego ya que además de tener un funcionamiento correcto se ha de poder manejar al personaje de la manera más cómoda posible.

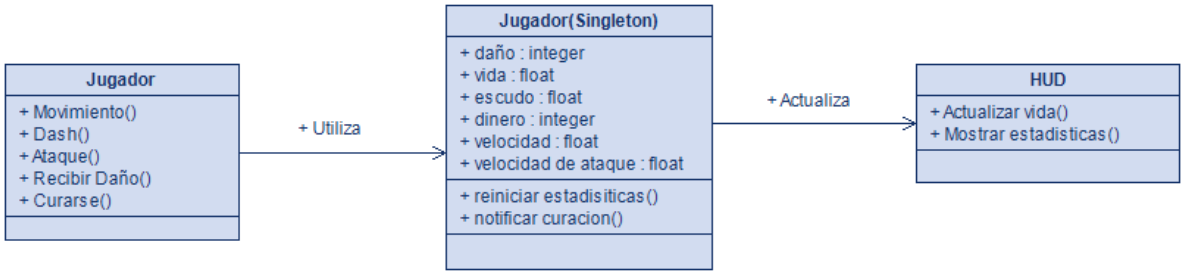

Figura 21

El personaje se mueve usando las teclas **W, A, S y D**. Según la dirección que el jugador marque, el personaje se orienta y comienza a caminar en esa dirección mediante. La velocidad del movimiento se ajusta para que no sea brusca gracias a la **interpolación lineal** para que el movimiento se sienta más suave y natural.

La **interpolación lineal** es un recurso matemático muy utilizado en los videojuegos que consiste en hacer una transición suave entre dos valores. pongamos de ejemplo que movemos algo del punto A al punto B: en lugar de que aparezca de golpe en el nuevo lugar, la interpolación hace que se desplace poco a poco, creando un movimiento más natural.

Este método se usa mucho en el proyecto, por ejemplo, para que la cámara siga al jugador de forma fluida, para que las cartas se animen con suavidad, o para que el personaje no se mueva de forma brusca.

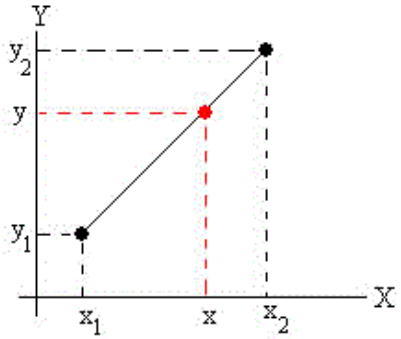

Figura 22

El personaje solo puede moverse si el juego lo permite en ese momento. Por ejemplo, puede haber situaciones en las que no se le deje caminar, como durante una animación o durante un diálogo importante como lo son los del tutorial.

Cuando el jugador se mueve, también tiene en cuenta los obstáculos que hay a su alrededor, para no atravesar paredes u otros elementos del escenario mediante su forma de colisión la cual está definida por un cuadrado a pesar de que en muchos otros juegos se utiliza una forma de cápsula.

Todo este comportamiento se actualiza cada frame dentro de **_physics_process(),** esta es una función de Godot que se ejecuta en cada frame sincronizada con la física del juego. Se usa principalmente para actualizar la lógica del movimiento, colisione.

El jugador tiene la capacidad de realizar un dash, este se activa al pulsar la barra espaciadora y permite al jugador moverse rápidamente en la dirección en la que se esté moviendo, ignorando ciertas colisiones.

El dash tiene una duración y un tiempo de recarga. Durante el dash, se desactiva el movimiento normal y se ajusta la velocidad con **interpolación lineal** para suavizarlo.

Al atacar, el jugador realiza un mini-dash, este ocurre automáticamente al atacar y empuja al personaje ligeramente hacia dónde apunta el cursor, simulando un avance rápido corto.

Ambos usan velocidad para aplicar movimiento directamente sin interferencias del teclado.

El sistema de combate del jugador también está implementado en código del personaje. Su funcionamiento se basa en ataques **cuerpo a cuerpo (melee)** que pueden encadenarse en un **combo de hasta tres golpes**. También incluye un **mini-dash** mencionado anteriormente que proporciona fluidez al combate y una mejor sensación al jugador.

Al atacar a un enemigo, este recibe daño gracias a las colisiones las cuales se gestionan mediante **grupos**, que permiten añadirles una “Etiqueta” para diferenciar las colisiones. En el juego hay tres grupos: **jugador**, **ataque** y **enemigos**. Si un enemigo colisiona con el jugador, este recibe daño o si el ataque impacta al enemigo, el daño lo recibe él.

A diferencia de un juego en 2D, la dirección del ataque puede calcularse simplemente con la posición del cursor en los ejes **X** e **Y**, pero en un entorno 3D se añade también el **eje Z**, que representa la profundidad. Para solucionar este problema, se utiliza un ***raycast*** que permite detectar la posición del cursor dentro del escenario.

Cuando el jugador ataca, se lanza un **raycast** desde la cámara del juego hasta la posición del cursor en la pantalla. Este raycast traza una **línea invisible**, devuelve el punto en el que impacta contra una superficie del mapa y seguidamente el área de colisión de ataque rota hacia la posición en la que ha impactado el raycast.

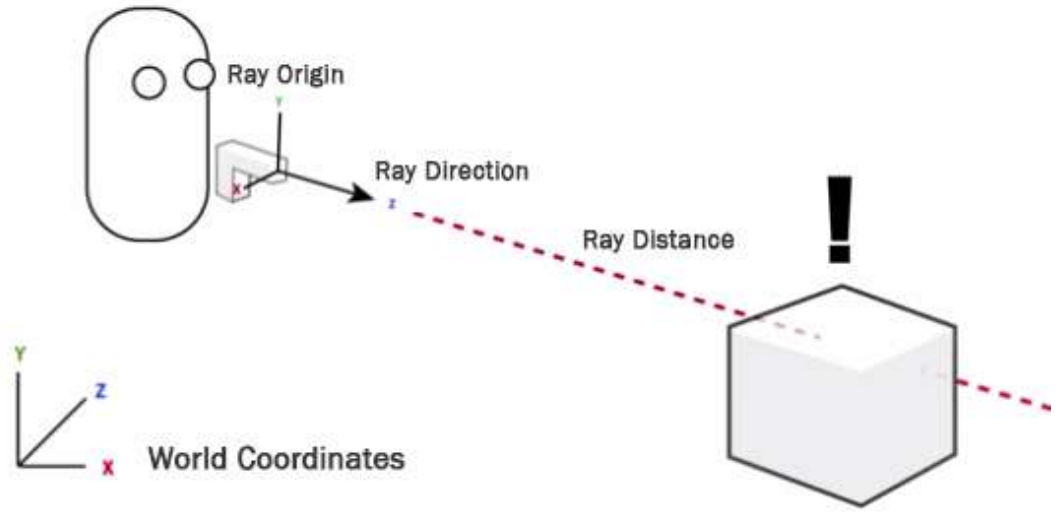

Figura 23

Por último, el personaje dispone de un sistema de animaciones en el que se utiliza el nodo *AnimationPlayer.* Con este nodo podemos crear animaciones complejas mediante el uso de unos puntos que se aplican a una línea de tiempo. eso nos permite hacer animaciones sin tener que hacerlas como se hace tradicionalmente.

En el caso del personaje, este está dividido en cinco partes: cabeza, bufanda, cuerpo, piernas y espada. Mediante el uso del nodo mencionado, se anima cada pieza del personaje por separado como la posición, rotación, o escala y luego juntándose en la animación completa, estos puntos funcionan indicando en la línea de tiempo que vaya de punto A a punto B en una franja de tiempo.

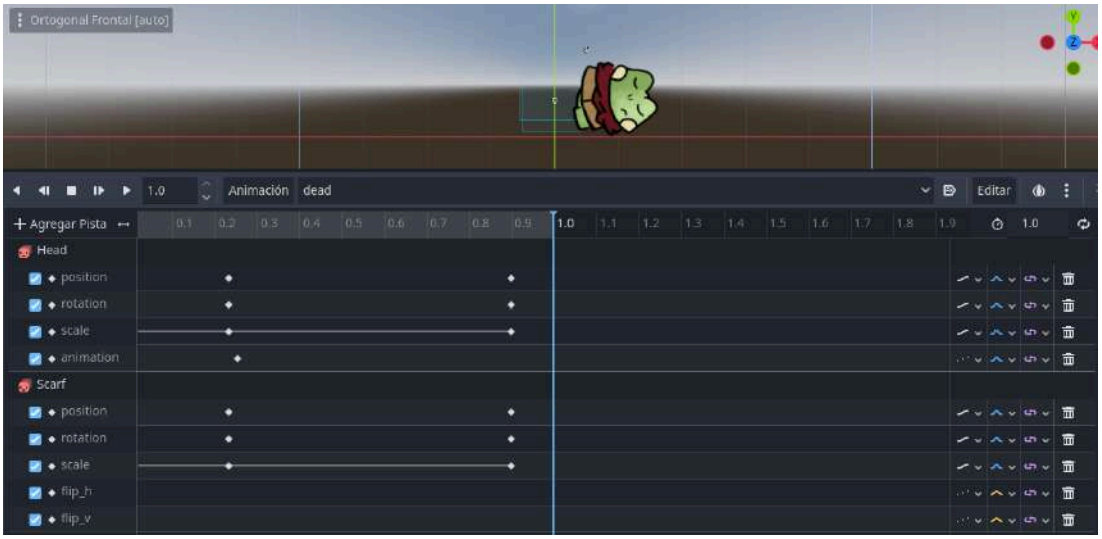

Figura 24

Sistema de Tienda

El sistema de tienda está diseñado para ofrecer una selección aleatoria de objetos al jugador, utilizando una **lista** de objetos disponibles, cada uno con propiedades específicas, como **nombre**, **precio**, **icono** y **probabilidad** de aparición. Además, cada objeto tiene un efecto asociado que se aplica al jugador al ser comprado.

La tienda cuenta con dos **altares** (Altar1 y Altar2) Los altares son una escena que se construye a partir del objeto aleatorio generado y adoptan este nombre ya que visualmente son como un altar con forma de tronco. Estos funcionan como puntos de compra independientes pero comparten el mismo código. Para mantener el estado de cada altar , se utilizan variables usando el patrón **Singleton** para determinar los objetos que se generaron y si ya fueron comprados o no.

Esto permite que al volver a entrar a la habitación, cada altar muestre correctamente su estado y el objeto asignado previamente independientemente del otro.

La selección de los objetos que aparecen en la tienda utiliza un sistema de **probabilidad ponderada**. Esto significa que cada objeto tiene una probabilidad de ser seleccionado, determinada por su valor de probabilidad.

Proceso de selección aleatorio:

1. **Suma total de probabilidades**:
   Se suman los valores de probabilidad de todos los objetos disponibles, obteniendo un total.
2. **Generación de un valor aleatorio**:
   Se genera un número aleatorio entre 0 y el total de probabilidades menos uno.
3. **Selección acumulativa:**
   Se recorre la lista de objetos, sumando de forma acumulativa sus probabilidades. El primer objeto cuya suma acumulada supera el valor aleatorio generado es el seleccionado que se mostrará disponible en el altar.

Este sistema de selección aleatoria funciona perfectamente acorde a este proyecto ya que ofrece una gran flexibilidad y adaptación de las probabilidades y ofrece situaciones más realistas.

En cuanto a los altares de las habitaciones de tesoro, funcionan de la misma manera que los de tienda solo que solo contamos con un “Altar” en lugar de dos y la lista de objetos disponibles se reduce solamente a los dos tipos de sobre con un precio gratuito.

Figura 25

Sistema de Items

Los ítems son una de las características clave del proyecto, ya que modifican las habilidades del jugador y mejoran sus estadísticas durante la partida. El jugador puede ir recolectando estos ítems los cuales están basados en cartas para conseguir llegar al final del juego con más facilidad.

El sistema de ítems, al igual que la escena *Room*, se basa en el uso de ***Resources***, aunque implementado de forma distinta. En este caso, el punto de partida es un script llamado **Item_Res.gd**, que extiende de Resource. Este script define las propiedades que puede tener un ítem, como su nombre, habilidades, tipo de carta, color, arte, entre otras.

Tal como se ha mencionado anteriormente, los *Resources* permiten crear **menús personalizados** dentro del motor a partir del contenido del script. Esta funcionalidad será útil más adelante, ya que nos permitirá crear los ítems de forma visual y eficiente.

ItemClass: actúa como la clase base para todos los ítems del juego. Es obligatorio que todos los ítems extiendan de esta clase para su correcto funcionamiento. Esta clase no se centra en la funcionalidad específica de cada ítem, como los efectos que aplican, sino que gestiona comportamientos genéricos compartidos por todos, como por ejemplo las animaciones, la detección del ítem seleccionado y el aspecto visual de la carta.

Para aplicar el aspecto visual en la carta, se accede a los nodos de la futura escena del objeto que representa la carta y se actualizan con los valores obtenidos del *resource*. Así, el texto del nombre, las habilidades, la imagen de la carta, su color de fondo y otros detalles se ajustan automáticamente según el contenido de los valores del *resource*.

Esta clase cuenta con animaciones mediante los *Tween* del motor. Este sistema permite interpolar valores como la escala, rotación o posición. Por ejemplo, al pasar el cursor sobre una carta, esta se amplía y rota ligeramente. Estas animaciones se llaman tanto al inicio de su aparición como al ser seleccionadas o pasar el ratón por encima.

Item Template: Es una escena que funciona como plantilla para facilitar la creación de ítems. Esta escena se encarga de mostrar visualmente la carta dentro del juego y de detectar si el jugador la selecciona. Esta escena incluye los nodos visuales necesarios para mostrar la información del ítem, como **Label** para los textos y **Sprite2D** para las imágenes.

Aquí es donde se exprime el verdadero potencial de los *Resources*. El nodo principal de esta escena tiene un script que extiende de **ItemClass**. Esta clase, al incluir un *Resource*, genera automáticamente un menú en el editor con los campos de este, permitiendo rellenarlos de forma visual como si se tratara de un formulario. Al ejecutar esta escena, la propia clase base se encarga de aplicar los datos correspondientes a los nodos de la carta, gestionando el aspecto visual de manera muy rápida.

Con este sistema crear un nuevo ítem es tan sencillo como duplicar la plantilla y cambiar los parámetros del menú por los de la nueva carta además de añadir su propio comportamiento.

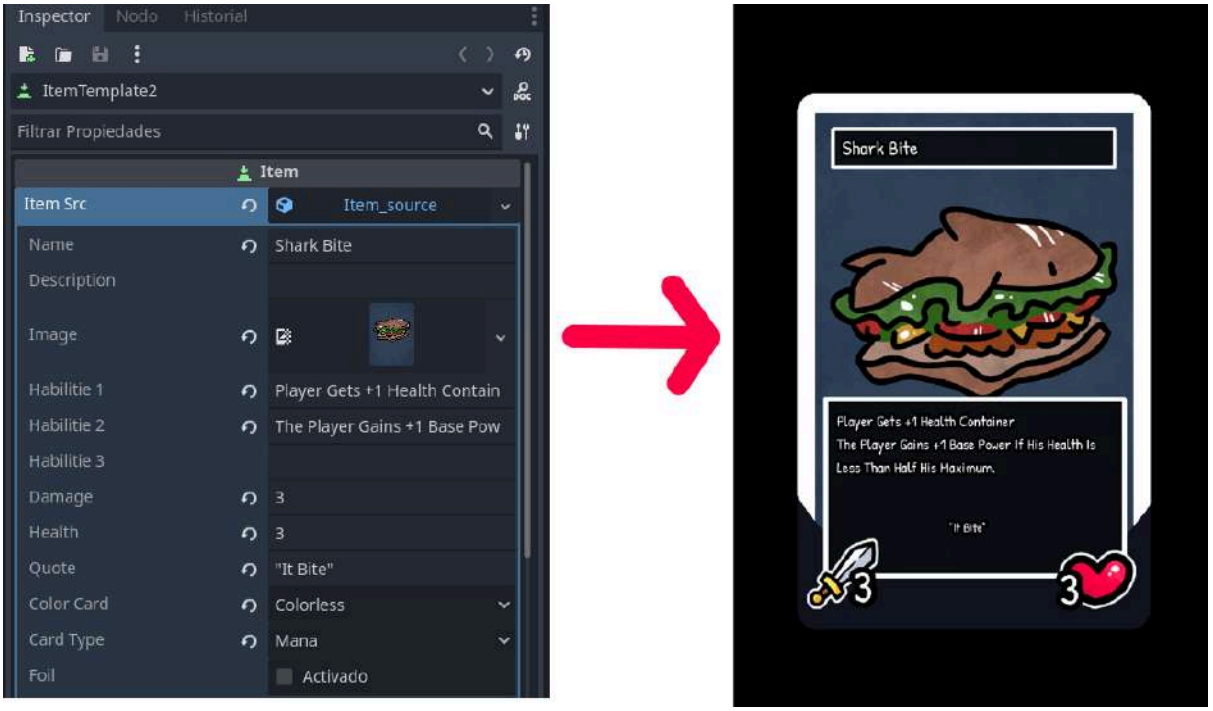

Figura 26

Apply Items (items.gd): Es una clase Singleton encargada de aplicar los efectos de los ítems sobre el jugador. Esta clase maneja la lógica de modificación de estadísticas, diferenciando entre ítems **momentáneos**(efecto inmediato) y **pasivos** (efecto que puede verse afectado a lo largo de la partida).

Para los ítems momentáneos, hay una serie de funciones que incrementan o decrementan directamente las estadísticas del jugador (vida, escudo, velocidad, dinero, daño base, etc.), aplicando los cambios sobre el Singleton del jugador

En el caso de los ítems pasivos, cada ítem pasivo tiene una función dedicada con su comportamiento. Al seleccionar un ítem pasivo, su función se añade a una lista. Esta lista se recorre en cada frame dentro del método **_process**, lo que permite actualizar constantemente el efecto de cada ítem de la lista.

Estas funciones se llaman desde el script propio de cada ítem. Lo ideal habría sido que el comportamiento de cada ítem estuviera dentro de su propio script, lo cual funciona para los ítems de efecto momentáneo. Pero en el caso de los ítems pasivos esto no es posible, ya que una vez seleccionado el ítem, la instancia se libera para optimizar el uso de memoria. Esto impide que el efecto pasivo se pueda manejar desde el propio ítem, lo que obliga a pasar la lógica a un script diferente.

Item Card Selector: Es la escena que se instancia al abrir un sobre de cartas, esta escena se encarga de seleccionar dos ítems aleatorios disponibles dentro de todos los del juego y mostrarlos al jugador para que seleccione una de las dos opciones.

Como hay demasiadas cartas en el juego, cargarlas por código puede ser algo que requiera demasiado esfuerzo y puede dar ciertos problemas de rendimiento. Para evitar esto, dentro de las carpetas del proyecto se han dividido los ítems por carpetas diferenciadas por el color de las cartas, por ejemplo, en la carpeta “Azul” se encuentran todos las escenas de items de color azul.

En lugar de cargar cada carta en memoria, solamente cargamos la ruta de las carpetas donde se encuentran las escenas de las cartas que podrán aparecer

Finalmente, se instancian las dos cartas que puede seleccionar el jugador. Para ello, necesitaremos obtener la ruta de la escena de la carta aleatoriamente.

Primero, se selecciona aleatoriamente un color de carta (por ejemplo, azul). Seguidamente, se accede a la carpeta asociada a ese color y se consulta cuántas escenas hay dentro devolviendo una lista de las escenas que hay dentro. Según esa cantidad, se elige una escena al azar dentro de esa lista y se obtiene su ruta completa. Esta ruta se utiliza para cargar e instanciar la carta y mostrarla en el selector.

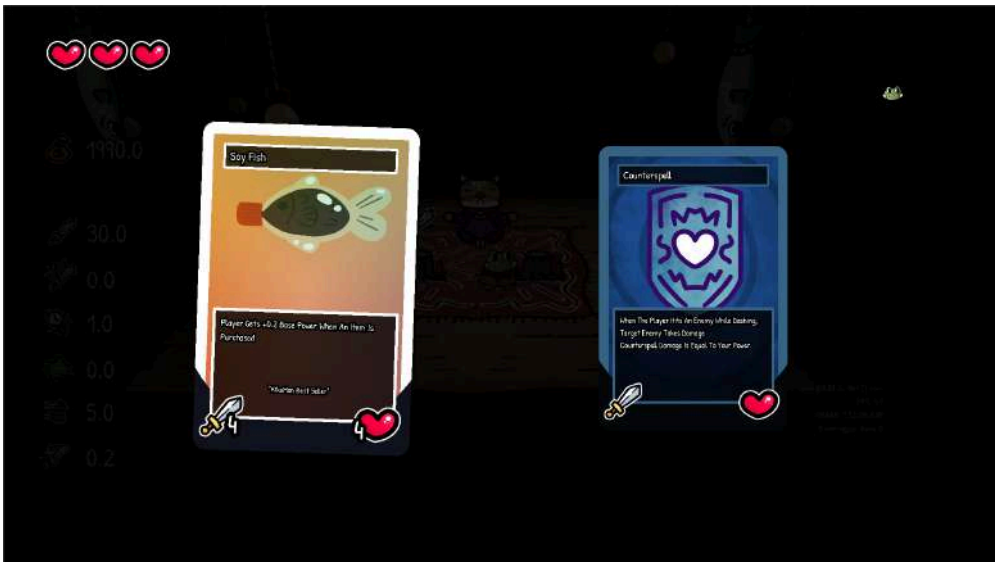

Figura 27

Comportamiento de los enemigos

El funcionamiento de sistema de enemigos, es muy similar a los métodos anteriores, para explicarlo de una forma general, al más bajo nivel los enemigos están compuestos por un *resource* llamado **EnemySource.gd** que gestiona las estadísticas generales que tendrán los enemigos (Vida, velocidad, retroceso…) Este resource al igual que el de los ítems contará con un menú propio en el motor para modificar manualmente estos datos.

EnemyClass: Esta clase gestiona el comportamiento base de todos los enemigos del juego. Se encarga de controlar varios puntos como recibir daño, generar recompensas al ser derrotados y moverse por el escenario en busca del jugador. Además, integra una lógica de máquina de estados sencilla que se usa o no dependiendo del enemigo.

La clase carga los datos desde del resource mencionado y define atributos como salud, velocidad y duración del retroceso al recibir daño. Además la vida del enemigo se modifica ligeramente en función del progreso del jugador para incrementar la dificultad.

Al detectar un ataque sobre el enemigo, el enemigo se le reduce la salud, emite en efecto de flash gracias a un *Shader* y se reproducen varios efectos de sonido. Si la salud llega a cero, se genera una recompensa aleatoria y se instancia un efecto visual en forma de polvo y se libera al enemigo de la memoria.

También se incluye un sistema de retroceso (*knockback*) que afecta su movimiento al recibir daño. El desplazamiento del enemigo funciona gracias a un nodo llamado *NavigationAgent* el cual estará dentro de cada escena de enemigo y permite que el enemigo siga al jugador dentro un área definida dentro de cada nivel.

Este método que permite a los enemigos seguir al jugador se conoce como ***pathfinding***. Es una técnica muy usada en el desarrollo de videojuegos, y su propósito principal es encontrar la ruta más corta entre un **punto A y un punto B**, evitando obstáculos del camino.

Por último, la clase cuenta con funciones que permiten añadir o quitar al enemigo de ciertos grupos de colisión para evitar recibir o hacer daño, así como aplicar un efecto de *flash* temporal cuando recibe daño.

Enemy Template: Funciona de forma similar al *Item Template*, pero con una estructura más flexible. A diferencia de la clase base de items, aquí no se accede directamente a sus nodos, permitiendo que cada enemigo pueda tener una escena ligeramente distinta según el tipo de enemigo, sobre todo si cuenta con múltiples partes visuales. Incluye los nodos que comparten todos los enemigos, como el *NavigationAgent3D* para el movimiento, colisiones y un *AnimationPlayer* para gestionar las animaciones.

Al igual que en el *Item Template*, esta escena cuenta con un nodo padre y tiene asignado un script que extiende de la clase EnemyClass. Este script utiliza el *resoruce* anterior para definir las propiedades del enemigo, como su salud, velocidad….

Gracias a este patrón, es posible crear nuevos enemigos fácilmente duplicando esta plantilla y modificando sus propiedades y comportamiento dependiendo del enemigo.

Para generar dónde van a aparecer los enemigos en la sala, se utilizan dos piezas fundamentales, una de ellas es el script *SpawnPoints* y la escena Spawner. El script SpawnPoint, implementado como un singleton, almacena una lista de patrones de aparición, que indican las coordenadas posibles donde pueden aparecer enemigos en cada habitación.

Por otra parte, la escena Spawner cuenta con un script que detecta automáticamente en qué habitación está y selecciona aleatoriamente uno de los patrones asociados a esa sala mediante el singleton SpawnPoint. Una vez seleccionado el patrón, por cada posición, se generará un enemigo aleatorio.

Los enemigos son los siguientes:

Slime: Este enemigo utiliza una máquina de estados sencilla definida en Enemy Class. Cuenta con tres estados principales: moverse aleatoriamente, quedarse quieto y atacar. Al iniciar, selecciona un estado al azar y al acabarlo selecciona otro hasta ser derrotado. Para moverse, elige un punto aleatorio dentro del mapa y se dirige hacia él. En el estado de ataque, primero se detiene y luego se dirige rápidamente hacia la posición del jugador.

Sardina: Este enemigo tiene el comportamiento más sencillo, su funcionamiento es muy básico y consiste en moverse constantemente hacia la posición del jugador a lo largo de la sala

Topo: Es un enemigo que puede ser bastante molesto, este, basa su funcionamiento en tres partes que se resuelven en este orden: Aparición→ Disparo en forma de “X” o “+” mediante unos vectores → se esconde y se mueve a una posición aleatoria repitiendo este proceso hasta ser derrotado

Enemigo “Piedra”: Este enemigo es el más molesto con diferencia, ya que cuenta con un área que detecta si el personaje está dentro o fuera del área. Si el jugador se encuentra dentro este se esconderá y no se le podrá atacar pero si el jugador sale de ese área se alejará del enemigo. este saldrá de la piedra y dispara tres proyectiles en forma de abanico hacia el jugador.

Rey Slime: Funciona de la misma manera que el Slime normal, comparten el mismo código solo que este va dejando un rastro pegajoso por el suelo el cual hará daño al jugador si pasa por encima

Búho Místico: Este jefe final cuenta con tres tipos de ataques distintos: invocar tres Slimes que luchan con el, lanzar una ráfaga de proyectiles en forma radial con un efecto de rotación desde su posición, o marcar zonas en el suelo que, tras unos segundos, dispara un círculo de proyectiles en todas direcciones. Al perder la mitad de su vida, entra en una segunda fase en la que se aceleran sus ataques para darle más dificultad a la batalla.

Persistencia de objetos y habitaciones visitadas

Persistencia de objetos: Es una de las características principales en los juegos del género *roguelike* es la persistencia de los objetos, si un enemigo deja caer un objeto al ser derrotado como una moneda, un escudo o un corazón, el jugador debe tener la posibilidad de volver más tarde a esa sala y encontrar el objeto. Para lograr esto, en el proyecto se ha implementado un sistema de persistencia de ítems que usa una clase global con patrón *singleton*.

Este *singleton* contiene una lista que guarda la posición y la habitación donde el enemigo soltó el objeto. Cada objeto que se añade a la lista contiene una clave generada a partir de las coordenadas de la sala dentro de la matriz (por ejemplo, "3,2"), y contiene una lista con los datos de cada objeto persistente en esa sala, incluyendo su tipo y posición.

Por ejemplo, si un enemigo suelta un corazón en la posición (4.2, 0.0, -1.7) dentro de la sala ubicada en las coordenadas 3,2, se almacenará en la lista indicando que en esa sala y en esas coordenadas existe un item de tipo corazón.

El sistema se basa en tres funciones principales:

1. **Añadir un ítem a la habitación actual**: cuando un enemigo suelta un objeto, este se registra en la lista, que guarda tanto el tipo del objeto como su posición dentro de la sala.

2. **Eliminar un ítem del registro**: cuando el jugador recoge el objeto (por ejemplo, al obtener un corazón), este se borrar de la diccionario y evitar que vuelva a aparecer en futuras visitas.

3. **Obtener ítems de la lista**: al acceder a una sala, consulta los ítems que contiene esa habitación y devuelve todos los ítems persistentes para esa sala.

A la hora de mostrar los objetos que persisten en la habitación, al entrar, el propio script de la habitación obtiene la lista de objetos que hay en esa habitación y las instancia en sus respectivas posiciones y al salir de esta los libera para que no aparezcan en la siguiente habitación ya que se deben mostrar solamente los objetos de esa habitación.

Habitaciones ya visitadas: Cuando el jugador derrota a todos los enemigos de una habitación, las puertas se abren y puede ir a la siguiente sala. En ese momento, al pasar por la puerta, se guarda la posición de la sala dentro de una lista en la clase global tipo singleton, y almacena todas las habitaciones ya superadas.

Al entrar en una habitación, el sistema comprueba si fue completada anteriormente. En el caso de que si haya sido completada, se eliminan automáticamente todos los enemigos y las puertas permanecen abiertas desde el principio, lo que permite al jugador avanzar fácilmente por zonas ya exploradas.

Sistema de Guardado

Poder guardar el progreso es muy importante en la mayoría de los videojuegos. En este proyecto se ha implementado un sistema de **autoguardado**, con el objetivo de que el jugador no tenga que preocuparse por guardar la partida manualmente.

El guardado automático se activa **al finalizar una partida completa**, lo que significa que no es posible guardar durante la partida. Esto se debe al diseño del juego y se inspira en títulos como *Cult of the Lamb*, que solo permiten guardar fuera de la partida, a diferencia de otros juegos como *The Binding of Isaac*, donde se puede guardar en cualquier momento.

Este sistema se basa en un script llamado **save_system.gd** que utiliza el patrón singleton, este script cuenta con tres funcionalidades básicas tanto para el progreso de la partida como para la configuración de los ajustes las cuales son: **guardar**, **cargar**, y **borrar.**

En cuanto al **guardado de partida**, el sistema recoge varias estadísticas del Script Global *Singleton* como por ejemplo el número total de enemigos derrotados, tiempo de juego, partidas completadas. Estos datos se almacenan en un archivo dentro de los siguientes directorios:

Linux: ~/.local/share/Froguenture/savegame.dat

Windows: C:\Users\Usuario\AppData\Roaming\Froguenture\savegame.dat

Al presionar el botón de continuar en el menú principal, estos valores se aplican en las variables singleton Global si detecta que hay un archivo de guardado.

Además, el sistema también gestiona **la configuración** del juego (por ahora: volumen general, música y efectos). Esta configuración se guarda automáticamente al cerrar la pestaña de configuración y se carga al iniciar el juego, permitiendo mantener la configuración del usuario.

Estos archivos se guardan con la extensión **.dat** en formato binario, lo que impide editarlos directamente con un editor de texto. Solo Godot puede interpretarlos correctamente, y los protege para que su información no pueda ser modificada desde fuera del juego.

Sistema de audio

Cuando un videojuego cuenta con un diseño de sonido coherente, transmite una mayor sensación de profesionalidad. En este proyecto se ha buscado ese efecto mediante una ambientación musical compuesta por tres pistas principales: la base, la de tienda y la de batalla. Estas no son canciones completas, sino capas instrumentales separadas que comparten la misma duración y están pensadas para reproducirse en bucle de forma sincronizada.

Desde el inicio de la partida, las tres pistas suenan simultáneamente, pero su volumen se ajusta dinámicamente según la situación del jugador. Por ejemplo, si entra en combate, la pista de batalla aumenta el sonido y se combina con la pista base dejando la pista de la tienda en un volumen prácticamente nulo, de otro modo, al entrar en la tienda entrara la pista de la tienda en combinación con la base. Este sistema de fundido de audio (fade) permite una transición fluida entre los distintos momentos del juego, evitando cambios bruscos ya que en realidad las tres pistas están sonando a la vez pero ajustando el volumen.

El juego también cuenta con diversos efectos de sonido *SFX* que se usan en momentos más específicos como al abrir un sobre, atacar y no requieren un sistema propio.

Para controlar el volumen de la música y los efectos de sonido, Godot ofrece una herramienta llamada **buses de audio**. Esta permite agrupar los sonidos por grupos y ajustar su volumen o aplicar efectos por separado. En este proyecto se utilizan tres buses: **Master**, **Música** y **Efectos**.

Estos buses están asociados a un script singleton llamado **Config.gd**, que almacena los valores de volumen de cada grupo. Desde el menú de opciones, el jugador puede modificar estos valores a través de *sliders*. Estos cambios se aplican de forma inmediata tanto al Config.gd como al volumen de los buses.

Figura 28

HUD

El HUD del juego es el encargado de mostrar de forma constante la información del jugador, como su vida, el dinero y otras estadísticas. Todos estos datos se obtienen a partir del singleton Player.gd, lo que permite reflejar cualquier dato de manera inmediata gracias a la función **_process** que comprueba la información en cada frame.

Para darle respuesta al usuario, se ha implementado un sistema que cambia el color de las estadísticas dependiendo si han bajado o han subido. Este sistema compara los valores actuales con los valores registrados anteriormente. Si detecta que un valor ha aumentado, el texto se pone verde por un segundo, por el contrario, sí ha disminuido, se muestra en rojo. Esto permite al jugador identificar rápidamente si el ítem que ha seleccionado ha subido o bajado sus estadísticas.

La escena **HUD** además transforma la vida del jugador de forma gráfica compuesta por corazones y escudos, los cuales son dos escenas independientes que solamente pueden cambiar de imagen. Este sistema se basa en tres variables: la vida actual, la vida máxima (o contenedor de vida) y los escudos.

Para cada unidad de vida máxima se genera un icono:

- Si el jugador tiene al menos 1 punto de vida, se muestra un corazón lleno
- Si tiene medio punto de vida, medio corazón
- Si no tiene vida, se muestra vacío.

Seguidamente, se añaden los escudos, que también pueden representarse como llenos o medios.

Todos los iconos se recogen en una lista y se ordenan de forma que primero se muestran los corazones (llenos, medios y vacíos) y después los escudos (llenos y medios) permitiendo así que para bajar la vida al jugador primero deberán quitarle el escudo.

Para mantener una interfaz limpia, se limita el número total de iconos visibles, y se colocan en dos filas, con un salto de línea a partir del séptimo icono. Finalmente, cada icono se instancia como una escena y se posiciona en la interfaz según su orden en la lista.

Los iconos que representan la vida y el escudo, solamente se actualizan visualmente cuando les llega la señal de que la vida ha sido modificada, una vez recibida esta señal, es cuando entra en proceso la actualización visual de estos.

Sistema de Diálogos

La escena **TextBox** maneja el sistema de diálogos del juego, la cual permite mostrar texto de forma secuencial con un efecto de escritura dentro de una burbuja de diálogo. Este sistema consiste en leer líneas desde un archivo **.txt** y mostrarlas dentro de la burbuja. Cada línea del archivo se procesa y se almacena en una lista que se va recorriendo conforme avanza el diálogo.

El diálogo se muestra dentro de una interfaz visual basada en un nodo del tipo ***NinePatchRect***, este representa la burbuja de diálogo y es un tipo de nodo que permite crear contenedores escalables sin deformar las esquinas. Este nodo es útil para los cuadros de diálogo, ya que se pueden adaptar a diferentes tamaños de texto manteniendo una apariencia visual. En este caso, el ***NinePatchRect*** contiene un nodo tipo Label donde se muestra el texto y a medida que se va mostrando el diálogo, la burbuja va adaptando el tamaño acorde a la longitud del texto.

A medida que el diálogo se va mostrando, cada línea se muestra con un efecto de escritura letra por letra haciendo que sea más dinámico. Este efecto se logra gracias a un temporizador que controla el intervalo entre cada carácter, generando así una sensación de que el personaje está hablando.

Además, con cada letra que aparece, se reproduce un pequeño sonido. A este sonido se le modifica ligeramente el **tono** en cada carácter para que el sonido no sea tan monótono.

Finalmente, para mostrar un diálogo, se añade esta escena al personaje que dirá el diálogo y desde el Script del propio personaje le indicamos a esta escena la ruta del diálogo a mostrar y la posición donde aparecerá.

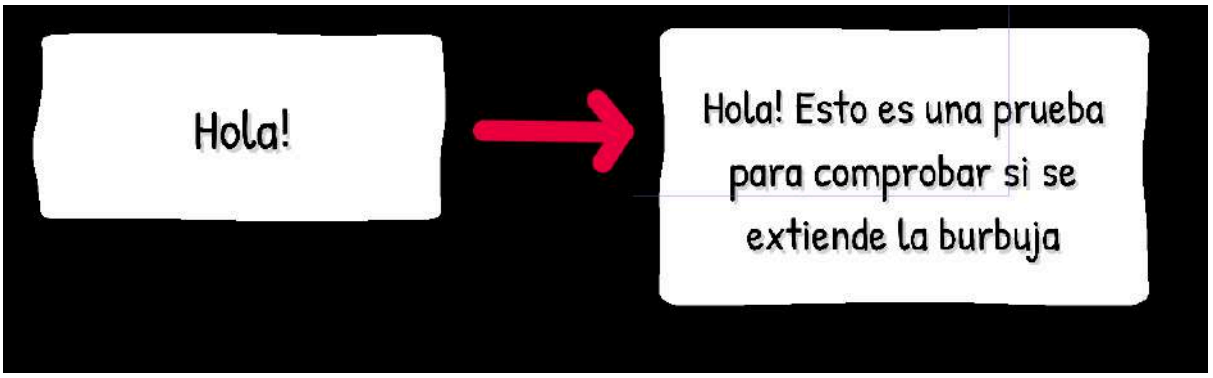

Figura 29

2.2.3 Implementación de shaders y materiales

En Godot, muchas escenas del juego incluyen nodos (como *Sprite2D*, *MeshInstance3D*, etc.) que tienen aplicado un material o un shader para controlar su aspecto visual.

Un material en Godot es un recurso que indica cómo se ve un objeto: su color, textura, brillo, transparencia…

Por otro lado, un shader sirve para aplicar una lógica a un material para crear efectos visuales personalizados, como:

- Agua animada
- Sombras dinámicas
- Cambios de color

En este proyecto se han utilizado estos recursos en muchísimas ocasiones pero las más destacadas son las siguientes:

Dithering Distance Fade: Este efecto es un material que ha sido aplicado a cualquier objeto que pueda tapar a la cámara dentro del juego. Este efecto se basa en calcular la distancia de la cámara a la del objeto e ir ocultando a este basado en la distancia de la cámara. Además este efecto para darle un toque más característico se utiliza la técnica de ***dithering*** la cual simula que el objeto se desvanezca por píxeles.

Figura 30

Wave Motion Shader: Este efecto igual que el anterior, se ha utilizado en muchas partes del juego. Este, está más enfocado para las interfaces. Podemos ver este efecto aplicado en varios lugares como: Menú de pausa o burbujas de diálogo.

Este efecto ha sido recopilado de la página de shaders de godot y funciona de la siguiente manera: Primero se calcula una forma circular con un qué borde varía con el tiempo, simulando un efecto de onda. Esta variación consiste en unas fórmulas matemáticas que generan curvas, lo que genera un movimiento en forma de ondas.

Figura 31

2.2.4 Dificultades y soluciones

A lo largo de este proyecto han aparecido muchos errores que se han ido resolviendo con el tiempo a lo largo del desarrollo. Algunos de estos problemas han tenido un impacto considerable en el proyecto, mientras que otros fueron prácticamente irrelevantes.

A continuación, se presentan algunos ejemplos destacados donde se encontraron dificultades y cómo se solucionaron:

Problemas con la cámara al finalizar la partida: Al terminar una partida y regresar al *lobby*, aparecía un error visual en el que el juego se mostraba desde una vista lateral en lugar de la vista frontal. Este problema estaba relacionado con las cámaras del juego, ya que estas se alternan entre la cámara 3D del personaje y la cámara 2D del mapa, activándolas según el contexto.

El error ocurría porque, al finalizar la partida, las cámaras no se gestionaban correctamente, y el juego seguía utilizando la cámara del mapa (la cual siempre está presente en la escena *Run)*, aunque solo se debería mostrar en momentos específicos.

La solución fue sencilla, se configuró la cámara del jugador para que tuviera siempre la prioridad por defecto, y únicamente se activa la del mapa cuando está visible. Antes de arreglarlo, Godot controlaba automáticamente la prioridad de la cámara del jugador, lo que por suerte solo ha sido un error en esta parte pero podría haber sido un problema mayor.

Figura 32

Problemas con la detección de habitaciones superadas: Este fallo fue bastante sencillo de resolver, y surge a partir del conteo de enemigos. Cuando un enemigo aparece se actualiza un valor el cual indica cuántos enemigos hay en una habitación, si este valor es cero se abrirán las puertas por que no hay enemigos, el problema es que este contador a veces indicaba que había un enemigo cuando no estaba presente en la habitación. Para solucionar esto, directamente pasamos de comprobar si no había enemigos en la habitación a que si la habitación estaba en la lista de habitaciones superadas omitiera lo demás, esto se añadió también en la escena que controla la aparición de enemigos (*Spawner*), si detecta que ya se superó la habitación se libera y pasa por alto el proceso de generación de los enemigos.

Problemas al exportar el juego: Al crear el ejecutable del juego surgieron un problema principal, este era que no se mostraban los diálogos dentro del juego. Esto se debe a que godot, al exportar el juego, excluyó los archivos **.txt** que contienen los diálogos del juego por que los detectaba como algo externo al motor. Para solucionarlo fue bastante sencillo se añadió la extensión .txt en una pestaña en la sección de exportación para que godot lo tenga en cuenta a la hora de crear el ejecutable.

Problema al mostrar el mapa: un problema que surgió durante el desarrollo fue que al mostrar el mapa por primera vez, no se muestra hasta que detecta cualquier mínima interacción del usuario, ya sea mover el ratón, pulsar un botón…

A pesar de haber probado diversas formas de arreglar este error como por ejemplo, forzar su visualización o redibujado, este error no se ha llegado a solucionar y por lo tanto queda pendiente para una futura actualización.

3. Resultados y pruebas

3.1 Videojuego en funcionamiento.

A continuación se muestran diversas capturas de pantalla mostrando diversas situaciones a lo largo del videojuego:

Figura 33

Figura 34

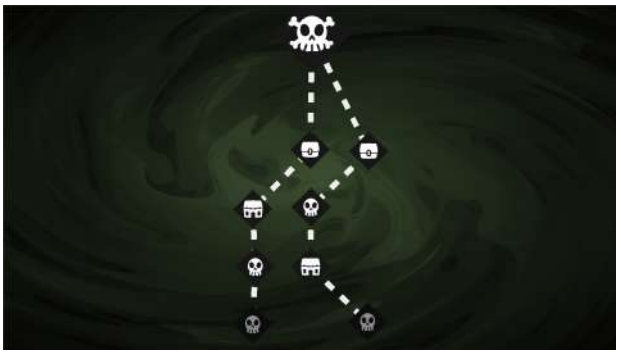

Figura 35

Figura 36

Figura 37

Figura 38

3.2 Pruebas de rendimiento

Para medir el rendimiento del videojuego, en una versión muy temprana se añadió en el hud un texto que servía para monitorizar el rendimiento del videojuego, a la par que se jugaba directamente al juego. Aquí se mostraban los frames por segundo y la memoria de video.

Más adelante, se utilizó una herramienta dentro del propio motor dentro de la pantalla de **Depuración**, En esta pestaña podemos seleccionar los valores que queremos controlar, mostrando unos gráficos que nos permiten ver este rendimiento de una forma más visual.

Figura 39

Una prueba de rendimiento que se utilizó para comprobar que tan bien puede manejar el juego situaciones extremas. En este caso llenamos una habitación con cien monedas para ver si habían bajadas de rendimiento al cojerlas todas de la manera más rápida posible.

Para esta prueba se plantearon dos situaciones que dieron resultados diferentes. En el primero, colocamos las cien monedas directamente dentro del árbol de la escena, lo que provocó que el rendimiento del juego se viera seriamente afectado y funcionará de forma muy lenta. En la segunda prueba, instanciamos las cien monedas por código, una por una. De este modo, el juego mantuvo un rendimiento normal. Solo se notó una pequeña bajada de rendimiento al recoger todas las monedas de golpe, pero fue prácticamente irrelevante.

3.3 Resultados de pruebas realizadas con usuarios.

Para este apartado se publicó el día 23 de mayo de 2025 una **versión de prueba** “Beta” para un público cerrado destinado a la **recopilación de errores**. El público seleccionado fueron tanto usuarios casuales como estudiantes de programación. En ambos casos se detalló que la versión disponible contaba con estadísticas alteradas y estaba planteado para la búsqueda de errores. En ambos casos los usuarios que probaron esta versión de prueba participaron en la búsqueda de errores, de los cuales un pequeño porcentaje reportaron errores que ya han sido solucionados, mientras el otro porcentaje, no llegaron a notificar ningún error pero sí propuestas o mejoras a futuro.

Durante la semana que se publicó esta **versión cerrada**, se han registrado 56 visitas y 36 descargas. Comparándolo con otros juegos publicados anteriormente, es una cifra considerable contando que tiene un acceso restringido.

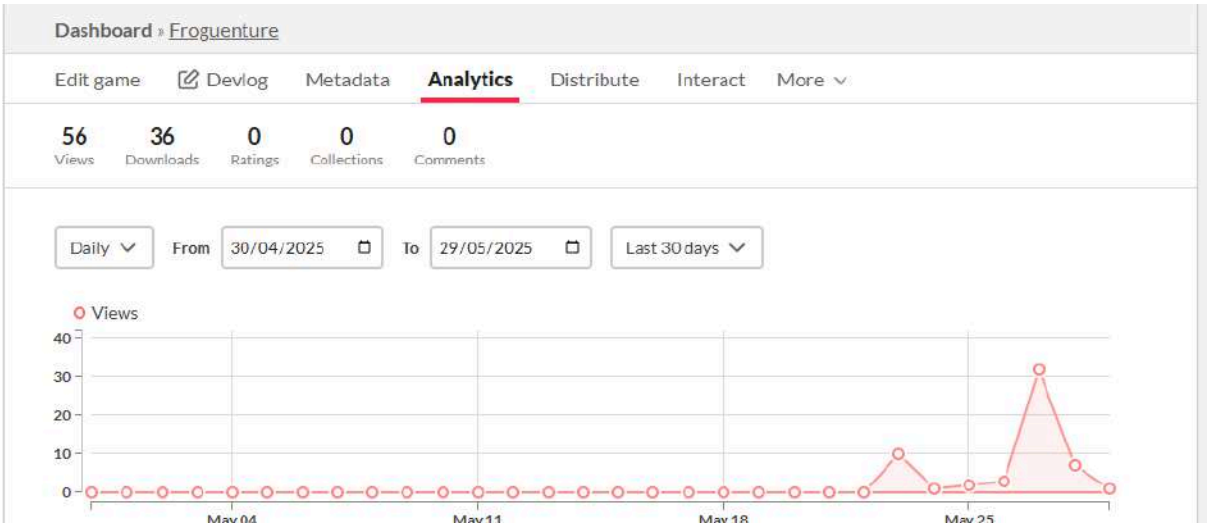

Figura 40

3.4 Respuestas y opiniones de los usuarios.

A lo largo del desarrollo, se ha ido publicando mediante redes sociales el proceso de desarrollo del juego principalmente en *Twitter* y *Tiktok.* Gracias a esto, he podido recibir respuestas e interacciones de los usuarios acerca del proyecto.

Como vemos en las siguientes imágenes, en *Twitter,* los usuarios se han interesado en el proyecto mediante “Me gusta”, compartiéndolo y comentando las publicaciones.

Donde realmente se ha notado un gran interés por el juego ha sido mediante la red social *TikTok,* En esta se publicaron varios videos donde se mostraba el avance del juego en cada video, siendo el último de todos donde más opiniones positivas y comentarios se recopilaron, llegando a la cifra de 40.100 visitas.

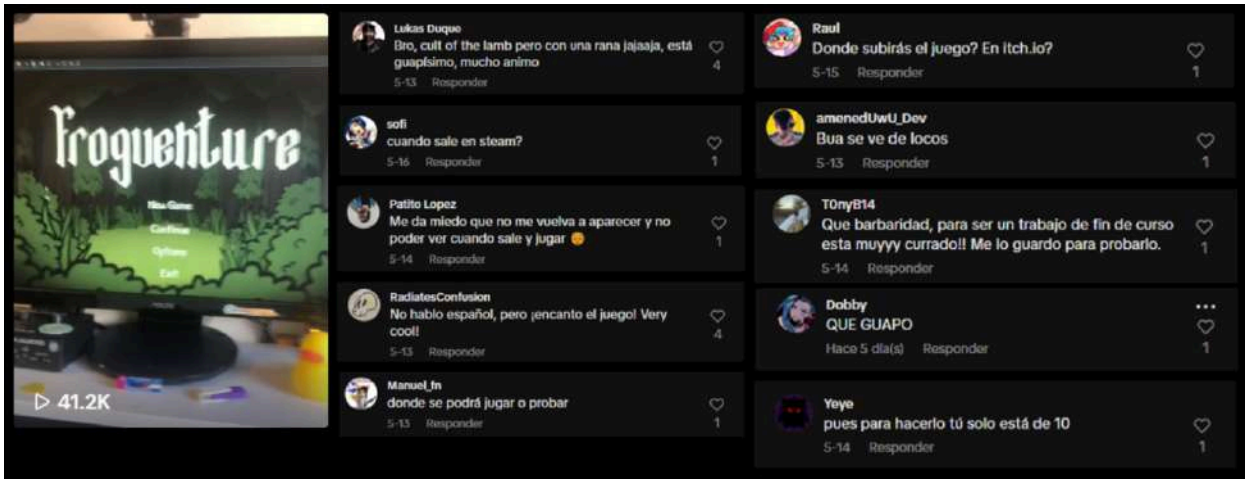

Figura 41

En general podríamos decir que el **90%** de las respuestas recibidas han sido positivas sabiendo que la mayoría de las personas son muy exigentes a la hora de jugar un videojuego y esperan una calidad muy alta indiferentemente del desarrollo que hay detrás.

4. Conclusiones

4.1 Conclusiones generales del proyecto

Para finalizar este proyecto, me gustaría mencionar que ha sido toda experiencia, el desarrollo de *Froguenture* ha sido un gran paso para adentrarme más profesionalmente en este sector, y entender las bases fundamentales de la generación procedural y como se trabaja en proyectos más elaborados cuyo desarrollo puede alargarse años.

A pesar de contar con una base bastante sólida gracias a proyectos personales anteriores relacionados con Godot y el desarrollo de videojuegos, he seguido teniendo la sensación de aprendizaje constante, quizás no a un nivel de un nuevo usuario, pero sí de conceptos más avanzados como la optimización de código o conceptos genéricos indiferente del motor relacionados con el desarrollo de videojuegos.

El tiempo de desarrollo se ha ajustado perfectamente al proyecto aunque muchos aspectos del juego me hubiera gustado trabajarlos más para un resultado todavía más profesional.

Como último punto me gustaría mencionar que todo lleva un proceso, el cual va acompañado de la paciencia y la constancia. Hacer un videojuego no es tarea fácil, y para mi este proyecto no es un juego cualquiera, es el reflejo de todo el esfuerzo empleado en el desarrollo y la muestra de que realmente soy capaz de cumplir mis objetivos.

Además gracias a este proyecto, me ha ayudado a enfocar los desarrollos de una manera más ordenada y plantear desde un inicio como funcionaran ciertas cosas y como pueden ser ampliadas en un futuro.

4.2 Objetivos superados

En cuanto a los objetivos que se mencionan al inicio, podríamos asegurar con certeza que se han cumplido con todos los objetivos propuestos, tanto para los específicos como los generales.

El juego se ha desarrollado con éxito y se han trabajado todos los puntos durante este proceso. Se ha conseguido un resultado profesional haciendo de este un proyecto con cara y ojos al público.

4.3 Visión de futuro

A este proyecto le quedan muchas horas de desarrollo por delante, en una futura actualización se añadirán nuevas funcionalidades y contenido para dar lugar a situaciones más aleatorias.

Entre las futuras características se encuentran:

- Objetos que modifican las estadísticas del jugador
- Habitaciones predefinidas nuevas
- Habitaciónes de eventos con personajes nuevos
- Sistema de generación a partir de semilla (para volver a repetir partidas)
- Enemigos nuevos

Muchos de estos puntos ya están en desarrollo pero han quedado fuera de la memoria y de la versión de prueba del juego por la duración del proyecto.

Además se intentará abrir el sistema de donaciones de itch.io para que cualquier usuario pueda contribuir mediante donativos y subirlo a la plataforma Steam mediante esos ingresos.

5. Glosario

Froguenture: Nombre del videojuego desarrollado en el proyecto

Roguelike: Género de videojuegos caracterizado por la generación aleatoria de niveles y una alta rejugabilidad.

Generación procedural: Técnica de desarrollo que permite crear contenido de forma automática mediante algoritmos, como niveles, mapas…

Drunked Walk: Algoritmo de generación procedural que simula el recorrido aleatorio a través de una matriz, utilizado en este proyecto para generar niveles en cada partida.

DAG (Directed Acyclic Graph): Estructura de datos en forma de grafo, utilizada para generar mapas verticales jerárquicos con bifurcaciones y caminos hacia el jefe final.

HUD (Head-Up Display): Interfaz gráfica que muestra información relevante durante la partida

Singleton: Patrón de diseño que permite crear una única instancia de una clase accesible desde cualquier parte del código. Usado en Godot para almacenar datos globales como estadísticas del jugador o configuración del juego.

Template: Patrón de diseño basado en una estructura base reutilizable. En este proyecto se usa para objetos como ítems, donde se parte de una plantilla común para crear elementos personalizados.

Item: Objeto que modifica las habilidades del jugador durante la partida. En Froguenture están representados como cartas de diferentes colores y tipos.

Resource: En Godot, es un tipo de objeto que permite almacenar y reutilizar datos.

Pathfinding: Técnica usada para encontrar el camino más corto hacia el jugador, evitando obstáculos.

Raycast: Técnica que consiste en lanzar un rayo invisible desde un punto en una dirección específica para detectar qué objeto o superficie impacta primero. Se usa comúnmente en videojuegos para detectar colisiones, seleccionar objetos o calcular direcciones.

Tween: Sistema de interpolación usado en Godot para crear transiciones suaves en movimientos o transformaciones de objetos.

Itch.io: Plataforma en línea donde se publican videojuegos independientes.

6. Bibliografía

Godot Docs – 4.3 branch. (s. f.). Godot Engine Documentation.
https://docs.godotengine.org/en/4.3/index.html

Acid Burritos – Grimbo Palette (s. f.). Lospec.
https://lospec.com/palette-list/grimbo

Emilio Moretti (2019, May 11). Godot Forum – How to stack 3d sprite so it respects the order?.
https://forum.godotengine.org/t/how-to-stack-3d-sprite-so-it-respects-the-order/23189

Godot Shaders. (2025, May 14). *Godot Shaders - Make your games beautiful!*
https://godotshaders.com/

Godot Engine Discord server (n.d.). Discord.
https://discord.com/invite/godotengine

Godot Café Discord server (n.d.). Discord.
https://discord.com/invite/zH7NUgz

James. (2022, July 4). *Slay The Spire: Map Generation Guide - KoSGames*. KosGames.
https://kosgames.com/slay-the-spire-map-generation-guide-26769/

GodotGameLab. (2024, February 21). *How to generate a slay The Spire-style roguelike map in Godot 4 (S02E05)* [Video]. YouTube.
https://www.youtube.com/watch?v=7HYu7QXBuCY

Vlad Govorov. (2021, October 25). *Map from Slay the Spire MADE EASY | Godot Tutorial | GDScript* [Video]. YouTube.
https://www.youtube.com/watch?v=dyfU-5Nbn_4

Queble. (2024, September 19). *How to add Screen Shake in Godot!* [Video]. YouTube.
https://www.youtube.com/watch?v=JDzUjJypQ9s

Kextex. (2022, September 10). *Distance Fade - Godot Engine 4.0 - Visual Shader* [Video].YouTube. https://www.youtube.com/watch?v=3XcxYFDi2Zs

Alberto Luviano. (2020, May 8). *Godot Game Engine - 3 ways to fade objects.* [Video]. YouTube. https://www.youtube.com/watch?v=OliCX6xEbUU

Matio888 . (2025, April 10). Freesound. https://freesound.org/people/llariio/sounds/797929/

HerbertBoland. (2007, April 12th, 2007). Freesound. https://freesound.org/people/HerbertBoland/sounds/33637/

FoolBoyMedia. (2016, January 3rd). Freesound. https://freesound.org/people/FoolBoyMedia/sounds/332323/

7. Anexos

Descarga del videojuego:
https://nacho-herrero.itch.io/froguenture

Repositorio del proyecto
https://github.com/NHerreroo/Froguenture

Generación procedural de los niveles:
https://github.com/NHerreroo/ProyectoFinCurso/blob/main/Scripts/LevelGenerator.gd

Generación procedural del mapa:
https://github.com/NHerreroo/ProyectoFinCurso/tree/main/Scripts/MapGeneration

Sistema de items:
https://github.com/NHerreroo/ProyectoFinCurso/tree/main/Items/_ClassResource

Personaje principal:
https://github.com/NHerreroo/ProyectoFinCurso/blob/main/Scripts/Player.gd

Esquema detallado del flujo del juego:

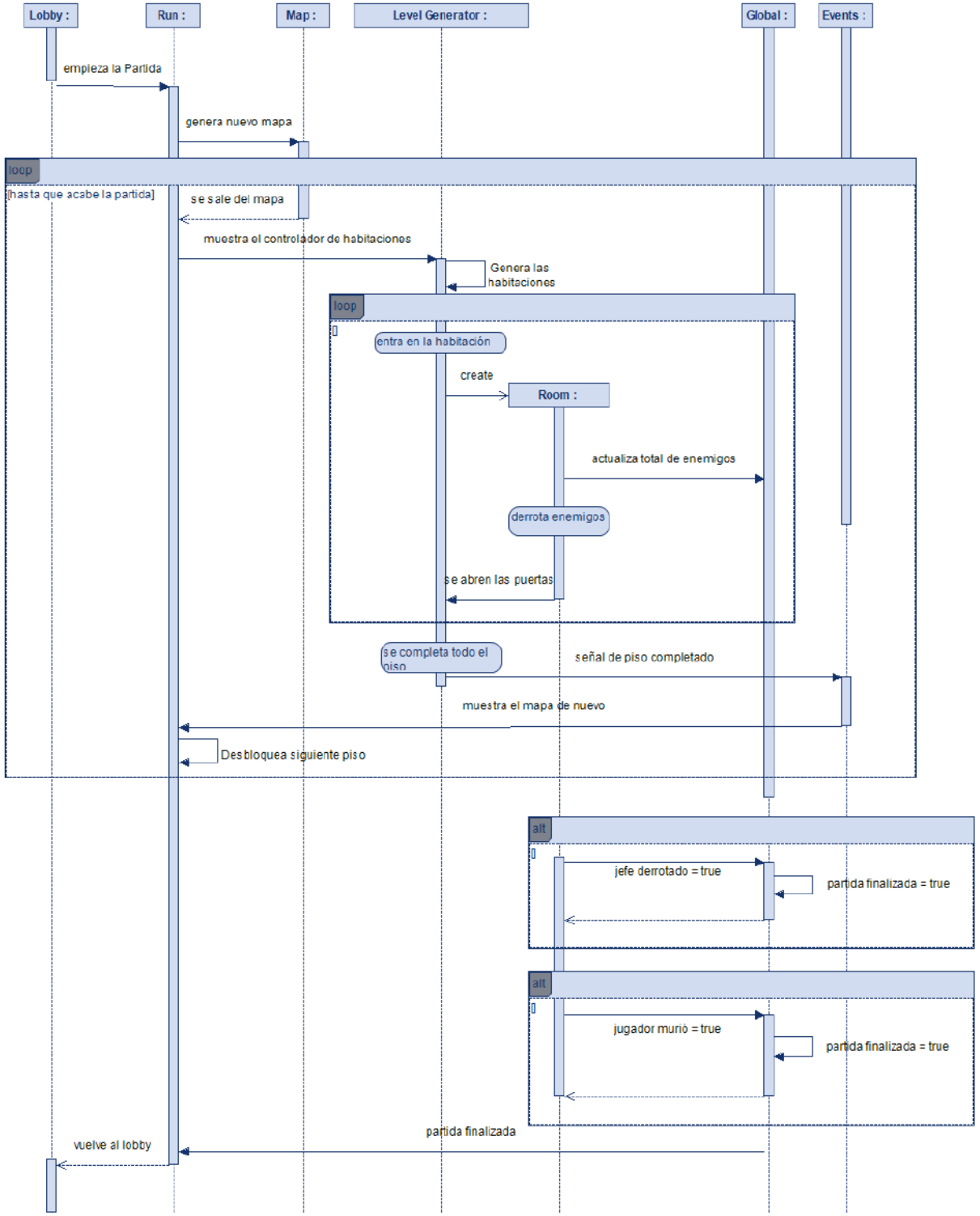

Figura 42