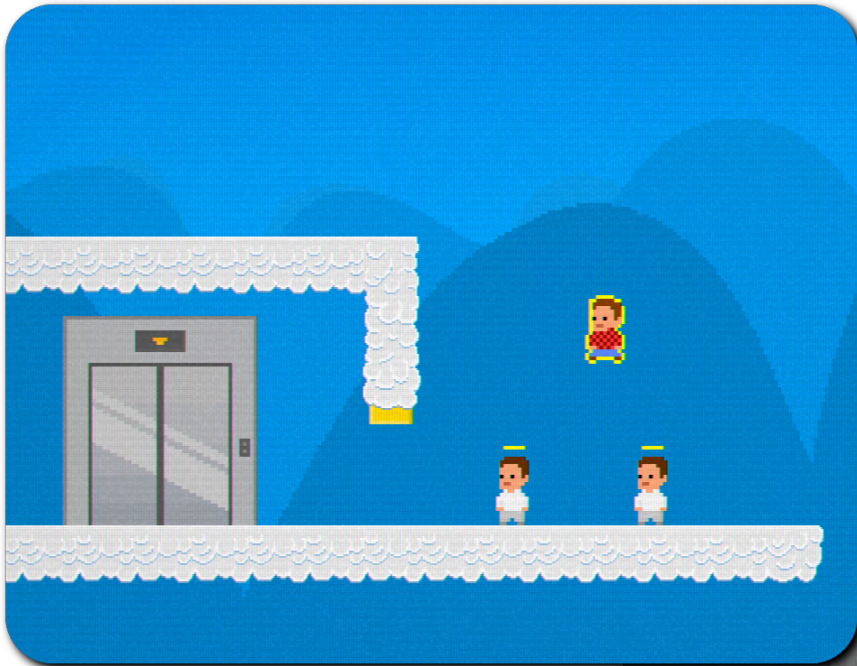


HEAVEN'S HELL



Development project

CFGS Desenvolupament d'Aplicacions Multiplataforma

David Estorach Pérez

DAM 2B

2024-25



INSTITUT
PUIG CASTELLAR

© (DecotDev)

Reservats tots els drets. Està prohibit la reproducció total o parcial d'aquesta obra per qualsevol mitjà o procediment, compresos la impressió, la reprografia, el microfilm, el tractament informàtic o qualsevol altre sistema, així com la distribució d'exemplars mitjançant lloguer i préstec, sense l'autorització escrita de l'autor o dels límits que autoritzi la Llei de Propietat Intel·lectual.

Abstract

The project is a video game with Heaven and Hell as the main theme, but not together, each in its own game world meaning it has two completely different gameplays. Although both are in 2D, all the gameplay, mechanics, point of view and set are distinct, where Heaven is a calm realm, focused on learning and improving movement skills, collecting seven emeralds while dodging threats through the journey, Hell is a world of chaos, focused on dealing with threats, but instead of dodging them, it is done spreading a bit of violence and shooting.

In spite of being two different worlds, actions done on one side will have an impact on the other.

The main objective of this creation is to achieve a fun game for both calm and action gamers with a good game feel, where the player can choose whether exploration or action based on what is felt more like to play. Last but not least, learn a lot about game development in this beautiful and painful path.

To achieve my objectives I'm learning a lot about the game engine through an infinity of tutorials, and searching the best ways to develop my ideas, trying to stay on a good game feel line, testing every game mechanic till the one that feels the best gets found.

In conclusion, the will is to create a game that feels good at any cost having two separate gameplays and worlds progressing through both of them.

Keywords

- Godot
- Videogame
- Platformer
- Shooter
- Heaven
- Hell
- Pixel Art

INDEX

1. Introduction.....	5
1.1. Context.....	5
1.2. Justification.....	6
1.3. Objectives.....	6
1.3.1. General objective.....	6
1.3.2. Specific objectives.....	7
1.4. Project planning and strategy.....	8
1.5. Work methodology.....	9
1.6. Economic and budget study.....	10
2. Project description.....	12
2.1. Functional requirements.....	12
2.2. Non functional requirements.....	12
2.3. Technologies.....	13
2.4. Project structure.....	18
2.5. Components description.....	20
2.5.1. Heaven components.....	20
2.5.2. Hell components.....	23
2.5.3. Global components.....	25
2.5.4. Assets components.....	27
2.6. Functionalities description.....	28
2.6.1. Angel Player.....	28
2.6.2. Demon Player.....	29
3. Conclusions.....	30
3.1. General project conclusions.....	30
3.2. Objective achievement.....	30
3.3. Planning and methodology valoration.....	31
3.4. Future vision.....	31
5. Glossary.....	32
6. Bibliography.....	34
7. Get the project.....	36
7.1. Get Godot.....	36

1. Introduction

The project is a Godot 2D game taking place in both Heaven and Hell, as it is the main theme. The point of this theme is having two different kinds of gameplay in the same game, separated in two worlds with completely different sets, but still attached together through story line and progress, so as the player takes part in both worlds.

On one hand there is Heaven, the first contact of the player with the game. An exploring and calm storytelling world of mystery, focused on 2D platformer movement mechanics and player skills taking part in an ancient angel's civilization while trying to collect 7 lost emeralds around the world.

The game perspective is side-on, as it is the one that fits the best. Although the aim of the movement skills is to generate movement freedom where it can be practiced and mastered to achieve *god movement*, that high level is not required to enjoy the game or finish it, it is more focused for those who want a challenge. Despite some game areas won't be a walk at the park, it will still be completely achievable by non experienced players with a bit of practice.

On the other hand there is Hell, an action shooter world with top-down perspective where massive violence is requested to handle enemy hordes. In this world, the character descends through Hell floors killing waves of enemies using guns and skills, while gathering in-game currency and materials that will be needed to upgrade the player.

Although the wave enemies will not be so strong, boss fights have a strong and deadlier enemy.

The main ways of joining both worlds are three: The Elevator: An unknown hidden elevator inside the Heaven fortress that just a few Heaven citizens are concerned about, and it is used to travel between Heaven, and Hell, including Hell levels. Storyline: some actions done in one world having effects on the other. Progression, where the player can exchange all he got during Hell runs at Heaven, being able to upgrade the player stats, its guns and the skills.

1.1. Context

The context of the project is video games development, specifically in the indie sector, a surging world growing every day and rising its popularity with each astonishing indie game released. But the truth is it is not a bed of roses, it is a cruel world crowded of solo developers and small teams starting a video game project on their own, developing it for years, full of dreams and expectations, and the reality is that most of them fail, not even reaching the lunch of the game after all the hard work, and if they do, they compete with big companies and other

indie developers on an infinity of weekly new game releases on an oversaturated video game market.

Although the disastrous context enshrouding the project, what motivates it the most is a couple things, one being will to learn and the same dream as every game developer has, resulting from a gamer passion,

Although the disastrous context enshrouding the project, what motivates it the most is a couple things. The first one is a common dream of indie developers, an inevitable result of an endless gamer passion since childhood, the eagerness to create your “*own video game*”. And the second one is the willingness to learn while doing it.

1.2. Justification

The roots of the project are not innovative, due to the fact that there have already been double playability titles launched through the years. Most of them featuring it just as a temporal part of a level, but there is one game that achieved it on a better way, *Hell is Others*, a top down extraction shooter featuring a 2D horizontal world, where this 2D world is closer to a disguised menu than a playable mode, as it is just a process to enter the main mode, but the player is constantly using it and it's more fun than just clicking some buttons between matches.

The innovative part of this project is creating a video game where both worlds matter the same and each one has a completely different gameplay and playability. Rather than having the main game mode and the other lacking in importance being just a secondary feature.

To pair the main point of the project, the set of each world are completely opposites, being Heaven and Hell, aiming to clarify even more the differences between both.

Although almost everything is already invented, the project could possibly be in a hidden area of a well known sector, searching for its own place.

1.3. Objectives

1.3.1. General objective

The general objective of the project is to develop a fun video game with two completely different gameplays and sets. This objective is organized dividing the game into two worlds, one for Heaven and the other for Hell.

Although achieving a playable game is the main objective, there are more general objectives for the project. To start with, developing a good game feel is one of those, as making both worlds' gameplay feel great while playing is a very important aspect. It is something that all the players will notice instantly after launching the game for the first time and it will accompany the whole playtime.

Another significant general objective is to design a game logic that engages the player to continue playing, where the progression and reward system is the key to bring that off on the project. Letting the player unlock the first rewards in a brief, those being simple but useful rewards and then, keep raising the rewards level but increasing the requirement to get them exponentially.

This next objective is a controversial one and it has been decaying every year in the video games sector, the optimization and game technical stability. Which would mean accomplishing a great optimized code free of errors, staying away at all cost from having inefficient procedures or abusing from every frame functions to avoid performance issues. Also it is really important to ensure the game does not crash on normal execution cases and runs stable.

Moreover, a more artistic point, create and design both worlds node sprites to get good looking graphics based on respecting their Heaven and Hell thematic setting the seal on the world's contrast.

Last, but not least, learn. As game development is full of aspects one of the objectives is to learn a lot from each involved.

1.3.2. Specific objectives

The specific objectives are a list of more technical goals than the generals, and their goal in common is to make those general objectives real, achieving them on the project.

As the whole videogame game is like two different games, this section is splitted into three parts, one for Heaven, another one for Hell, and the last one for every common global objective, appearing in both games or in menus.

Heaven objectives:

- Angel player with good movement
- Heaven like world
- Interactive NPCs
- Collectibles
- Dangers
- Shop

Hell objectives:

- Shooting demon player with skills
Demon upgrades
- Wave based game mode
- Enemies
- Boss fights
- Collectibles

Global objectives:

- Main manu
- Pause menu
- Sound and music system
- An elevator to travel between worlds
- Specific GUIs for each world
- Good game feel

1.4. Project planning and strategy

It is known that there are different ways to face the development of a video game project of this size. Considering a couple strategies, one of those is following a complete project tutorial, upgrading it and finishing it with a personal touch. On the other hand there is the second way to confront it, building piece by piece the whole project from scratch.

The chosen strategy is the last mentioned. Although the first one would be an easier and convenient way to carry out the project, there are a bunch of reasons why to avoid it and go for the second and harder strategy. Being both the project idea and objectives motives enough, this is due to the fact that the game idea would not easily fit on an existing project and would get in the way of the learning process as a lot of steps would have been skipped. Luckily, starting from zero solves all that even though, brings a couple more stages which involve more work to do.

The project planning is divided into the following five stages:

Godot learning is the first phase. Learning the basics of game development and the possibilities offered by the game engine to be able to value more accurately the game idea and how to develop it.

Analysis and design is the second one, which consists in once having a base of Godot knowledge, start thinking about the game itself, how to drive the project idea into achievable objectives.

Prototype development is at third place, which englobes developing the game idea on a functional level, a prototype of each world free of artistic design.

Test, fix and upgrade is the next step. Moving from the prototype to a more serious development starting to create some final designs on the tested parts and having the code fixed.

Finish the game. The last step is to complete all the remaining objectives and polish every game aspect.

1.5. Work methodology

The project is being carried out by only one single developer, so the work methodology is completely flexible and does not require any kind of task distribution, indeed it is just a tool to have all the tasks stored and managed while creating or planning an adaptable work timeline.

So the methodology that suits the project situation the best, after considering a few, is Kanban, which is a task visualization tool based on a board with three important sections to organize the tasks state, those being the TO DO, DOING and DONE.

Moreover, not having to set a time limit like a sprint as other work methodologies do, in this case might do, as there is no past projects reference to evaluate how much time each task requires to be accomplished, which could result in a bit of planning chaos but, on a bit more efficiency.

1.6. Economic and budget study

This study represents the budget for a project of this size in a situation where the workplace is the developers home.

The price approximations are from real price data, ensuring a realistic approach to the budget.

ONE TIME BUDGET	
Laptop	700€
Graphic designer	800€ x 3
Sound designer	800€
Steam page	100€
Total	3200€
MONTHLY BUDGET	
Developer	2100€ gross
Internet	20€
Electricity bill	90€
Total	2210€ / month

Table 1. Relation of items by cost table.

The table below is created distributing the one time payments based on the different project steps accordingly.

MONTH	ITEMS	BUDGET
1st	Laptop	2910€
2nd	Graphic designer	3010€
3rd	-	2210€
4th	Sound designer	3010€
5th	Graphic designer	3010€
6th	-	2210€
7th	Graphic designer Steam page	3110€
8th	-	2210€
Total		21680€

Table 2. Monthly project budget.

2. Project description

2.1. Functional requirements

- Play
- Elevator to travel between worlds
- Playable characters movement
- Player gun: shoot, reload
- Player skills: dash, heal
- Collectibles
- Interactive NPCs
- Text bubble
- Enemies
- Levers
- Doors
- Pause
- Background music
- MP3 player
- GUIs
- Quit

2.2. Non functional requirements

- Game feel improvement
- Game performance
- Animations for players, enemies, objects, acciones, menus
- Sound effects
- Background music
- Automatic save and load progress from local file
- Enemies round based waves
- Maps design
- Backgrounds
- Tile maps layer tiles
- Animated scene transitions

2.3. Technologies

Game engine:

It is the main technology used on the project, where everything is built up from. A game engine is basically a game development environment or a framework prepared to create video games with integrated essential game logic components such as physics, rendering, camera and a lot more to make game development more bearable.

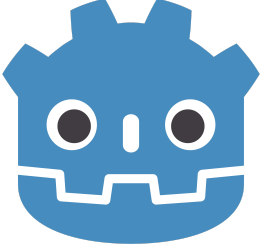
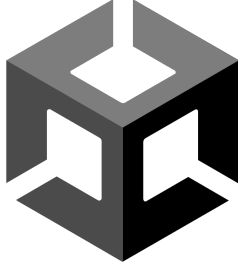

	Godot 	Unity 	Unreal Engine 
Ease of learning	Beginner friendly	Moderate	Complex
2D	Designed for it	Good	Not optimal
Coding languages	GScript, C# (C and C++ with extensions)	C#	C++
Community and tutorials	Growing and friendly	There are tutorials of everything	Moderate and not for beginners
Documentation	Great and updated	Average	Average
Price	Free and open source	Free if less than \$200K USD of revenue and funds raised in the last 12 months	Free, royalties apply after \$1M USD gross product revenue

Table 3. Game engine comparison table.

After shuffling and comparing the three game development platforms there is clearly one that just suits perfectly for this project situation, it is **Godot**.

First, a bit about Godot. It can be imagined as a theater stage builder, it works with **scenes**, highlighting a main one where everything happens, with some actors and set decoration taking part. On the game engine everything is done using **nodes**, existing a huge variety of them with their parameters and signals, where a player and a map obstacle of the scene would use two different types of nodes, both running inside a scene. It also uses tree hierarchy, inheritance and composition, enabling the option to create more complex actors by combining nodes on a separate scene and then using that scene inside the main one.

So, why Godot? There are some reasons, the first one is its ease to learn, counting with a beginner friendly learning curve, being also flexible and not requiring a very complex start to get the first things running on a brand new project. Additionally, it is encouraged by coding too, due to the fact that it has developed its own programming language, the **GDScript**, inheriting all the beginner friendly qualities.

Second reason is that out of the shuffled options it is the best one for a 2D game, as it was mainly developed with that aim, so that is another thumbs up to why bring the project blueprints on this game engine.

Another good motivation is its documentation, tutorials and community, always willing to help. It is true that on one hand there is a lot of outdated non official tutorials on the network, teaching for older Godot versions, but, on the other hand if something changed like could be the location of a specific checkbox needed to follow the tutorial, trends to be easy to find it on the newest version. Moreover, if that is not the case, there is usually someone who has already contributed giving details on the comments section about how it is done after the update. It is important that it also gets a network dedicated to sharing assets full of new options and plugins.

And things do not just end there, as it is an **open source** project, leading to more community implications such as frequent bug fixes, regular content updates and all the range of advantages given by this license.

In spite of **Godot** fitting perfectly the project need, **Unity** fitted great too but lacked from some aspects, not to mention that **Unreal Engine** was early discarded due to its focus on 3D projects, being short on the 2D scope. So what finally left aside Unity was all the restrictive clauses it owns, the fact that it has a bigger learning curve and a lot of functionalities on the game engine that actually will not be needed for this kind of project, instead would just disturb, making the learning experience more complicated and slow.

Pixel art program:

In 2D video games almost everything visual works with sprites, an image or a group of them assigned to the player or any game object to let it just be visible on the screen or even create any kind of animations. The sprites should follow some rules as not having a background and exported using PNG format, or if it is a sprite sheet for an animation all the frames ought to be the same size and position to avoid a lot of nuisances while using it on the game engine. So in order to create and draw the sprites using the correct format, a program that allows those functions is required.




	Aseprite 	Ibis Paint X 	Flipaclip 
Designed for	Pixel art and sprite animation	Digital painting and illustration	Animation & frame-by-frame drawing
Ease of learning	Beginner friendly	Moderate	Beginner friendly
Animation creation	Onion skinning and timeline	Manual with layers	Onion skinning and timeline
Platform	Windows, Linux, macOS	Android, iOS	Android, iOS
PNG export	Yes	Yes	Yes
Price	20€	Free version with ads or subscription	Free limited version with ads or subscription

Table 4. Pixel art drawing program comparison table.

Although not being the ideal program for pixel art sprite sheets, the chosen drawing application is **Ibis Paint X**.

The principal cause of this choice is the developer's familiarity with and knowledge of the application, gained through personal experience, which confirms that all the needed functions for the intended task are available and perform just great.

Even though **Aseprite** offers a better variety of tools designed for pixel art and easier to use, its price is what rules it out, as Ibis Paint X free version comes with almost every single tool and the blocked ones are able to be unlocked for a short period of time by just watching a single commercial advertisement.

Another good point on choosing a mobile application is that the work can be done everywhere and anytime, providing a lot of freedom and flexibility while the artistic part of the project is worked.

Digital audio workstation:

A DAW is a software designed mainly for music creation, but in this project it will be used with the purpose of creating not only music but also game sound effects. To achieve this the help of the digital synthesizer **Vital** is required.

Vital is a free and powerful wavetable synthesizer with high levels of flexibility and high-quality sound design features, allowing the modification of waveforms at a very precise level.





	FL Studio 	Ableton Live 	LMMS 
Designed for	Music production, SFX, loops	Live performance & sound design	Electronic music & SFX creation
Ease of learning	Advanced	Advanced	Moderate
Vital Synthesizer compatibility 	Yes	Yes	Yes
Export modes	WAV, MP3, OGG, FLAC, MIDI	WAV, MP3, AIFF	WAV, MP3, OGG
Platform	Windows, macOS	Windows, macOS	Windows, Linux, macOS
Price	99€ to 539€	79€ to 599€	Free, open source

Table 5. Digital audio workstation comparison table.

The chosen DAW is **LMMS**.

The decision was made for its outstanding **price** range compared with its direct competitors. As the project sound productions do not require a superb professional work, LMMS will do more than enough. Once it has been discovered that it works completely fine with the useful **Vital** synthesizer and lets export on MP3 and WAV audio formats, there is no need to consider any of the paid options, tossing out **FL Studio** and **Ableton Live** from the list.

If the price was not enough, there is more good news, because this program is **open source**, as it was originally designed for linux and nowadays it is free to use and enjoy, as all its license benefits on all common desktop platforms.



Figure 1. LMMS and Vital Synthesizer screenshoot

2.4. Project structure

The project structure works using a **tree hierarchy** of scenes and nodes with attached scripts. There are three scenes that work as a game screen, which can be called **running scenes** and can only run one at once, being the **current scene**. Those three being the **Main Menu**, **Heaven** and **Hell**. The last couple are more complex than the first one as they represent the game worlds, containing a lot of nodes and other scenes.

Then there are the **specific scenes**, that instead of acting like a world, they form part or they are in that world. Usually made out of nodes to achieve a modular world component, but they can have modular components too, having other scenes instantiated on them. A great example of this would be the player scene, it is instantiated on the world scene. But if for some reason the player needs to have a gun, instead of creating it inside the player node, it is better to use a modular way, creating the gun on a separate scene and then instantiating that gun scene in the player tree hierarchy. That would lead to the capability to create, for example, other guns and then add it to the player by just replacing the other gun instance with the new one.

Although the game could be developed on just a single scene containing all the nodes, that would be really inefficient and slow to create, as everything would be instantiated and running at the same time, and more importantly it would not be taking any kind of advantage of Godot

composition and modularity. Forcing the developer to build every game step by step or copy and paste nodes with all the references problems that conduces to.

The following diagram shows the project components till one layer inside the world scene, all ordered with its group color. It also shows the player flow through the game scenes, using the global common components when it is required, it is important to point out that almost all the **Global** group members are **static**, meaning they are accessible from anywhere within the whole project and they do not get restarted when changing the current scene.

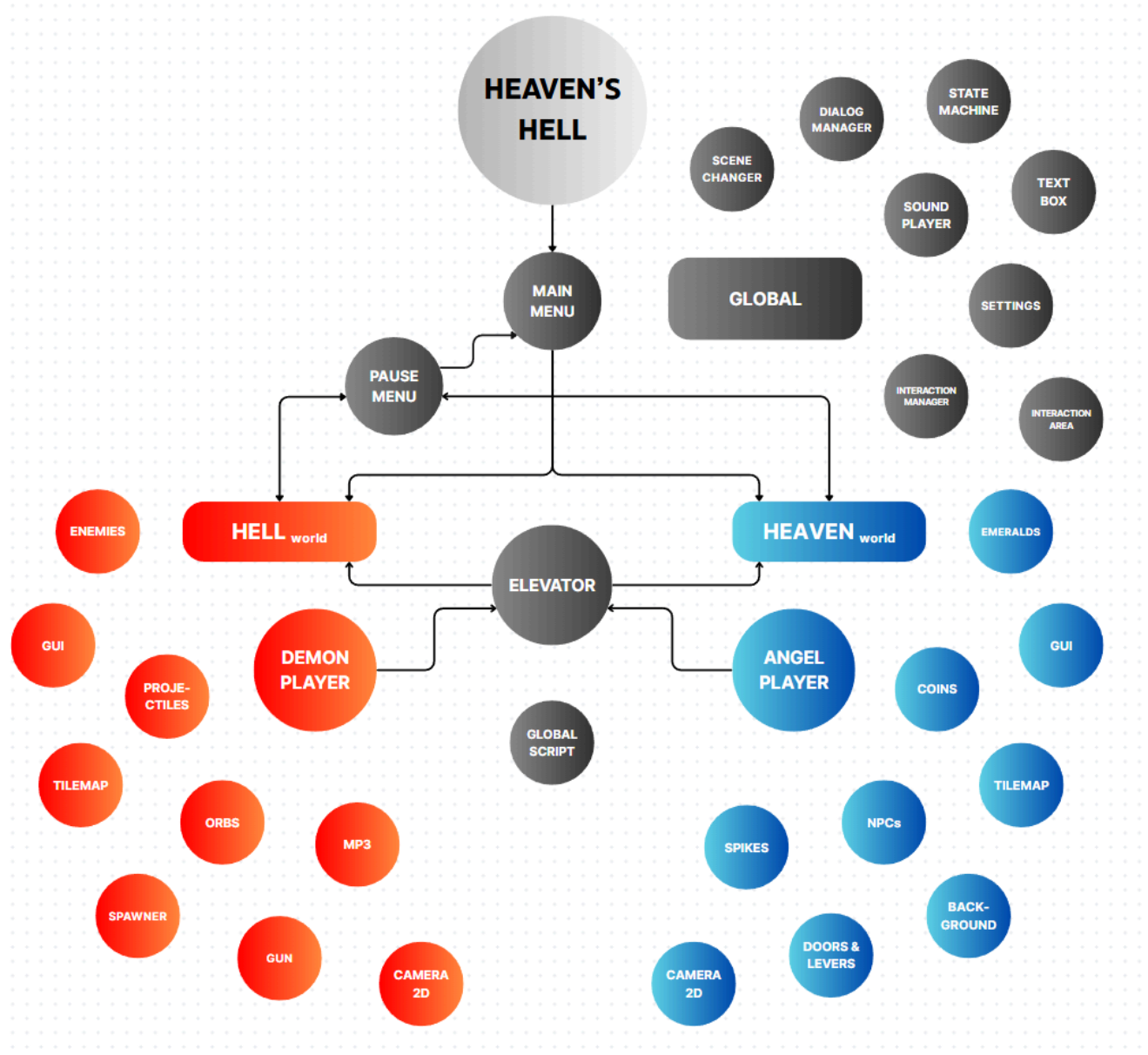


Diagram 1. Project components structure and player flow.

The next diagram shows the assets components used on the project, distinguishing three different groups. **Sprites**, which are the images of every texture based visual node. The **audio** group, including all sound effects and music. **Shaders** group, containing texture based code modifications. And lastly there is the **other** group, having the fonts.

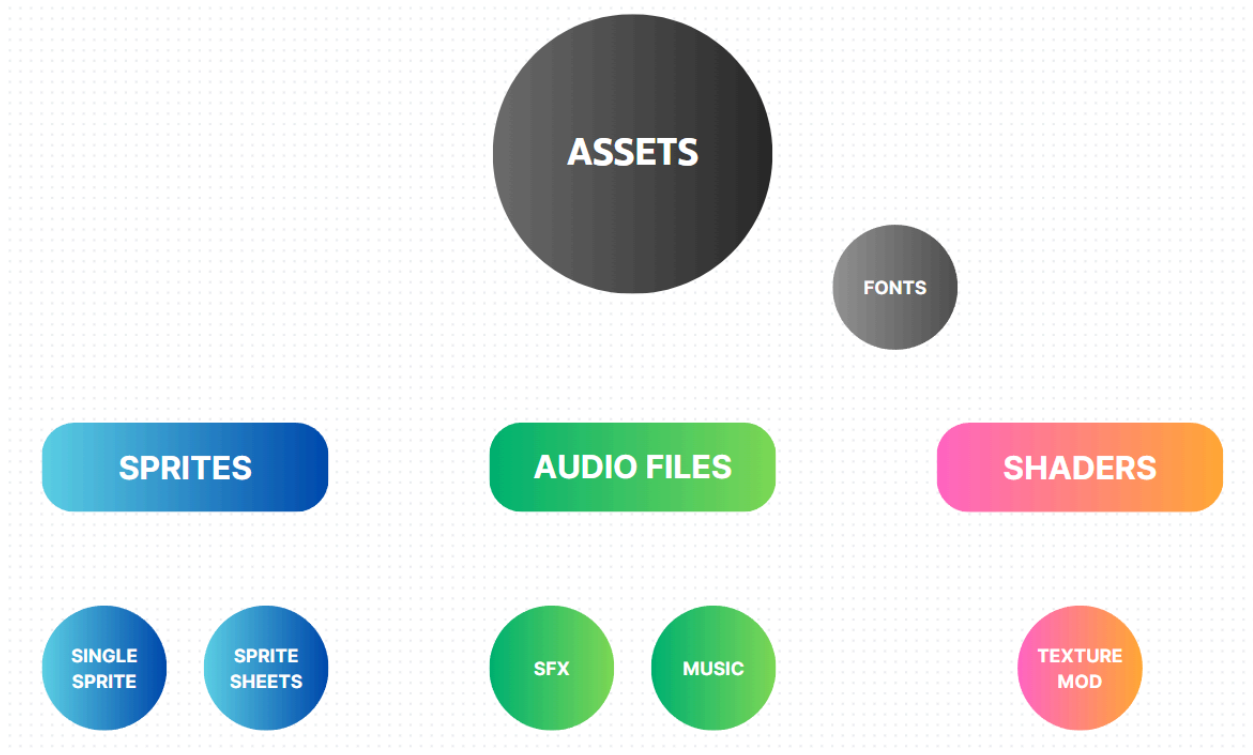


Diagram 2. Project assets structure.

2.5. Components description

Like the game and almost everything in this project, this section is divided into parts too, aiming to achieve a better distribution and understanding. Starting with the higher tree hierarchy elements, those being all scenes.

2.5.1. Heaven components

2.5.1.1. World

The heaven world is the place where almost every heaven component is instantiated, but not forgetting its own child nodes, so it does not require any special behavior, it is a node of type **Node2D**.

All the instantiated heaven scenes as children of this component correspond to the *Diagram 1*. above. Here is a list of their child nodes:

- **Parallax background:** A node prepared to create the **parallax** effect on the background, using one or multiple **ParallaxLayer** nodes, each with different speed configurations and having a **Sprite2D** node as children containing the layer sprite.

The script attached to this world node handles some start configuration methods from the global group.

2.5.1.2. Angel Player

Also named under *Lumber*, with a node type of **CharacterBody2D**, being the controlled character by the player. Its movement is handled with an advanced node based **State Machine** (explained below on the global group), it is remarkable that it counts with **coyote time** and **buffered jump** to get a better player experience.

It has its own script storing all kinds of variables, statistics of the player, control ones and connection references to its children nodes. Moreover the function that lets the player get hurt is defined here, instead of at the state machine code. This is done to get an extra level of reliability when called.



List of children nodes:

- **HeavenCamera:** The **Camera2D** node that follows the player position.
- **Sprite2D:** The default **Sprite2D** node for the character sprite sheet.
- **Respawn:** A **Sprite2D** for the respawn animation sprite sheet.
- **CollisionShape2D:** The default **CollisionShape2D** node for the character, prepared to collide with the world environment and collectibles.
- **StateMachine: Node** to attach state machine state scripts on children, here counting with 6 states: Idle, Running, Jumping, Falling, Dashing, Hurt and Respawnning.
- **Hitbox:** An **Area2D** to redefine the player hitbox with a different **CollisionShape2D**.
- **Timers:** A **Node** containing **8 Timer** nodes used on the player behavior.
- **AnimationPlayer:** A node of type **AnimationPlayer** used to create and play all the character animations using the sprite sheet frames and modifying other values.

2.5.1.3. Tile Map

A node of type **TileMapLayer** containing a cloud sprite sheet with a total of **47** hand made tiles painted with collisions, z index values and a terrain set to enable the **autotile** function.



Figure 2. Heaven Tile Set with painted autotile.

2.5.1.4. Graphical User Interface

This being a **Control** node featuring all the heaven world GUI items, having all its children on a **CanvasLayer** node to ensure the items never leave the screen area. On its children it can be found a couple of **Label** nodes to display the collectibles, an **AnimatedSprite2D** for the coins icon, a **Timer** for the emeralds label and an **AnimationPlayer** to handle elaborated items animations. On its script all the labels update functions are handled.

2.5.1.5. NPCs

With the same node type as the player, the NPCs are intractable **CharacterBody2D** nodes with a sprite sheet and animations handled by an **AnimationPlayer**. Using the **Interaction Area**, **Interaction Manager**, **Dialogue Manager**, **TextBox** from the Global group and its own script functions is what makes them able to be interacted with by the player, creating chat bubbles. For correct display they use functions attached to **Area2D** nodes signals.

2.5.1.6. Collectibles

Here three types can be distinguished, **Coins**, **Special Coins** and **Emeralds**. All three being an **Area2D** shaped by a collecting range collision, an **AnimatedSprite2D** and an **AnimationPlayer** for more elaborate animations. Each with its script reacting to the area signals and getting the GUI instance to call the needed label updates.



2.5.1.7. Spikes

A node of type **Area2D** able to damage the player when it notices its collision through the area signals. Being goldy, shiny but sharp thanks to its **AnimatedSprite2D** the player must avoid them not to get killed

2.5.1.8. Doors and Levers

These couple components work harmoniously together, while the **Lever** is a **Node2D** with an **InteractionArea** that uses a link code to attach the doors that are under its control, the **Door** nodes are **StatycBody2D** type to be able to block or give in the way to the player. Both have animated sprites and collisions.

An interesting point about the Door is that it uses an **AudioStreamPlayer2D** to reproduce the open and close sound positionally.

2.5.2. Hell components

2.5.2.1. World

The main scene for the Hell world, containing all the pieces to create its top down 2D shooting waves based game mode. The children of this node, are a couple of **Node** nodes to store other instantiated scenes in an organized way and an **Area2D** that acts as a start button when the player enters the arena square. Its script attached handles start configuration methods from the global group scope.

2.5.2.2. Demon Player

This node of type **CharacterBody2D** is the main character of hell, being the playable one, allowing the player to run, dash and shoot enemies. Its movement is handled by the **State Machine** with the states of Idle, MovingShooting and Dash. But, the other half of the gameplay is controlled by the **Gun** scene. This scene is visually controlled by a couple **Sprite2D** nodes and an **AnimationPlayer** also counts with three extra **Area2D** nodes handling its hitbox and orb pickup range. The character default collision manages the movement around the map.



2.5.2.3. Tile Map

A simpler tile based map layer for the floor, walls and background of the hell world. Hand made and with different **collision** and **z index** paint on the tiles creates a playable arena with defined bounds.

2.5.2.4. GUI

This important **Control** node is responsible for letting the player know its character statistics such as health, bullets, skills cooldown, orbs and more. It counts with a scene of a Hell **MP3 music player** that makes the background music a more customizable option for the player.

2.5.2.5. Camera2D

This **Camera2D** node attached to the **Demon Player** position and affected by the **mouse position** too gives the player a smooth shooting experience on the game.

2.5.2.6. Spawner

This **Node** type node, with some spawn point defined on the map, works as an enemy generator and wave control system, knowing how much enemies generate depending on the wave number and when to go by the next wave.

2.5.2.7. Enemies

The enemies are a crucial part of the game, existing two, **Flying Head**, a horned demon face that rushes with random movements to the player position, and a Temple. Although both are controlled using the **StateMachine**, the temple has a more complex movement pattern, searching for the player and when it reaches a minimum shoot distance starts shooting projectiles at the player.



Both enemies count with a variety of animations going from spawning, moving, attacking and once killed.

2.5.2.8. **Projectiles**

Projectiles are an **Area2D** node that depending on their parent could be bullets and kill the enemies or, hit the player trying to make him lose.

2.5.2.9. **Orbs**

This hell precious collectible appears from defeated enemies and can be picked up by the player by just stepping near it.

2.5.3. **Global components**

2.5.3.1. **Main Menu**

The main menu is the **first scene** runned when the game is launched, with an animated title text and some texture based buttons it lets the player choose which game mode to play, or close the game.

2.5.3.2. **Scene Changer**

This **static** component works as a tool to change the current scene with pre-made animations created with a couple of **ColorRect** nodes with the help of **shaders** and an **AnimationPlayer**. Its purpose is to make scene changes a bit better than just the new scene popping instantly on the game screen.

2.5.3.3. **Pause Menu**

This scene made out by a **Control** node, some **TextureButton** nodes, a background and an **AnimationPlayer**, although **not** being **static**, is instantiated on both game worlds, allowing the player to completely pause the game at any time. Moreover it is the way to come back to the main menu scene.

2.5.3.4. **Global script**

This script is designed to take complete advantage of the **static** quality, storing a lot of game variables needed across both worlds or even at the save and load files, because this file not only stores them, it also counts with specific methods to **save and load** those values into the progress save files.

2.5.3.5. Elevator

This node is an **Area2D** elevator designed to allow the players to travel between both worlds. In spite of its own area focused on controlling the **automatic elevator doors**, it uses the **InteractionArea** to do the interaction action to enter it, and with the help of a couple of **AnimationPlayers** it also is capable to play the elevator close and open scene transition animations.

2.5.3.6. State Machine

The **State Machine** is present at almost every moving character in the game, despite **not being static** it works by attaching the main state machine script to a node on the character scene. It requires at least one state to be functional, in order to create a state, a specific character **state template** to inherit is required. This template also inherits from a base state template.

On the character it results in a **StateMachine** node containing a child node for each character state.

The reason behind using a state machine is the high level of control that it gives under the character, letting it perform the actions on separate states with all the advantages given by the code isolation, **avoiding nested conditional** code lines.

2.5.3.7. Sound Player

The sound player is a **static** scene of a **Node** type that acts as a **collection** of **AudioStreamPlayer** nodes and a single **AudioStreamPlayer2D** with a variety of sound playing functions to satisfy any audio need. It also, on its own script, gets preloaded and saved on a variable, resulting with all the game sound effects and songs being able to play at any time.

2.5.3.8. Text Box

This scene is not static, it is a **MarginContainer** node containing a **NinePatchRect**, a node that with the specific texture given, allows to create a resizable box that always maintains its aspect ratio. Adding a label as the cherry on the cake with some script animations results into a great video game text box. It is only instantiated by the **DialogManager**.

2.5.3.9. Dialog Manager

Being the second **static** kind script on the project, is the responsible component of creating and destroying the text boxes, it also manages the text lines, allowing the player to navigate forward every message the game has for him.

2.5.3.10. Interaction Area

This node being an empty **Area2D** waiting to get a **CollisionShape2D** when implemented, works as an interaction button using a **Callable** function to register and connect the desired parent object action. As an area it registers the player collision adding or removing it from the **InteractionManager** area list.

2.5.3.11. Interaction Manager

The **Interaction Manager** is the one in charge of managing the interaction areas and players input. If there is more than one area it decides which is nearer the player, limiting the interactions to only one, avoiding double actions with one key. This works using the **areas list** it counts with, and some methods that use the players position.

2.5.4. Assets components

2.5.4.1. Sprites

Sprites are single frame images used on almost all visual different parts of the project. Every single visual texture based piece, coins, interface icons, and interaction prompts are built using these assets. They are used inside **Sprite2D**, **AnimatedSprite2D**, GUI nodes or every parameter that requires a texture.

2.5.4.2. Sprite sheets

Sprite sheets are multiple frame images used to create any kind of animation on characters, collectibles, enemies, or even at GUI nodes, basically everywhere that uses a texture based animation. They are also used on **AnimatedSprite2D** and **AnimationPlayer** nodes to create and play sequences through the premade animations.

2.5.4.3. **SFX**

Sound effects are files of **.mp3** and **.wav** format played in reaction to character actions, world or GUI interactions. All these are preloaded into the **SoundPlayer** scene at game launch and can be played from any script using the static Sound Player scene of the project.

2.5.4.4. **Music**

Background music is essential on video games, so this asset is really valuable. It plays depending on the scene the player currently is.

2.5.4.5. **Shaders**

This excellent but difficult to use tool has the purpose to modify the **texture based nodes** creating amazing effects or even overlays.

2.6. Functionalities description

2.6.1. **Angel Player**

- **Movement:** The player can move the character with freedom horizontally running and vertically jumping with different jump heights or descending faster.
- **Dash:** Once on air time, a dash can be performed with a given direction to travel more horizontal distance, at the end the player has a decreasing speed inertia. If it is done with no horizontal movement, the dash turns into a small vertical float.
- **Coyote time:** If the player falls from an edge just right before jumping, there is a reserved time where it would be still able to perform a jump airborne. This is done with a timer, providing a great game feel on tricky *parkour* situations, preventing a common gamer fall, feeling that the game is not responding fast.
- **Buffered jump:** Once the player is falling a jump can be stored for some milliseconds to just right after landing automatically jump again. This improves the game feel on multiple jumps in a row, to prevent the player from reacting slowly and falling too easily.

2.6.2. Demon Player

- **Movement:** The demon player moves horizontally and vertically equal thanks to the Hell world's top down viewpoint.
- **Change fire mode:** To swap between gun firing mode there is just required one press of the designed key, [B], pressing again the button resets the firing mode to the default one.
- **Shoot:** Using the left mouse click, the player is able to aim and shoot the enemies. The gun counts with a couple different firing modes, being full auto the default one and semi auto the secondary one. On the default mode, sustained left mouse click keeps shooting with a default rate of fire. While using semi auto there is no fire rate limitation but every shoot is done by a new mouse left click.
- **Reload:** There are multiple ways of reloading, using its designated key or continue pressing the shooting mouse click with the empty magazine till the gun starts reloading automatically.
- **Dash:** A speed boost dash can be performed, giving the player a small invulnerability boost and allowing him to dodge or escape from no escape situations. This skill has a cooldown timer.
- **Heal:** A heal skill can be used too, recovering a small amount of health and requiring a recharge to use again.

3. Conclusions

3.1. General project conclusions

At the end of this challenging learning path, the conclusion is that the project was directly disproportionate with the developer's will to create, its knowledge and personal values that were totally contradictory to the due date. Time goes fast.

The thirst to create a complete full doble video game from scratch, has been blinding this young developer from reality since the very first time that idea landed on his mind, resulting and exposing a wrong handled inexperience. Regardless of the fact that during the last development weeks, the blinding thirst started to fade, it was hopeless, as the time faded even faster.

However, desperate, the developer tried to avoid the obvious, and so cuts were made into that dream idea, so as to the project, hurting directly the developer's desired dream. Leaving a man, with an endless project on one hand and a broken heart on the other, hit hard by the reality of time and greed itself.

Mistakes were made, the first one being too ambitious trying to fit a big project on a short term development, the KISS principle was clearly not borne in mind. In second place, in an attempt to reduce the first time issues and the home brew flavor, a project restart was suffered, which resulted in more development delay.

The resulting project is a plenty of good tools to develop the game, but in its actual state is more like a playable demo with not much to do and repetitive.

Even though it is not a project focused on what the most job offers request for, and it was known since the beginning, the hard work has really paid off when referred to programming and Godot knowledge, as it has ridiculously grown thanks to the enthusiasm and the owned great programming base.

It also helped growing other virtues such as pixel art design or music and SFX production using a DAW, both also from scratch.

3.2. Objective achievement

When referring to if the project objectives were achieved, it is difficult to quantify in a realistic way, the short answer is, they have not.

It is true that there are a ton of accomplished objectives, but it is also true that during everyday development new objectives just kept appearing, even before finishing the already defined objectives, and the developer just did not want to discard them, even after hearing all the recommendations about shortening the project.

3.3. Planning and methodology valoration

Although the project officially followed the Kanban methodology, the reality is that the implementation turned out to be far from the desired. Kanban's simplicity and flexibility were appealing from the beginning, especially for a solo developer. However, as the project advanced, this same flexibility became a double edged sword. The rise of new ideas, mechanics, and visual improvements appeared constantly. And instead of sticking to the original plan, most of them were instantly added to the task list and developed, moving the project away from its end again and again.

Planning and sticking to a less flexible idea could have been a valuable option to solve this important issue.

3.4. Future vision

The future vision of this project is to try to carry it out the nearest as possible to the first idea, developing all that had to be cut by the time limitation, adding different levels, floors, shops, skills, story, bosses, player upgrades, better visuals, game feel improvement and a lot more!

Aiming for a more fun and playable Heaven's Hell version with a release on itch.io website.

5. Glossary

Node: The basic building block in Godot. Everything in a scene is a node or a combination of nodes. Nodes have different types like `Sprite2D`, `Area2D`, or `AnimationPlayer`, each with unique functionality.

Scene: A scene in Godot is a collection of nodes organized in a tree hierarchy. Scenes can be instanced into other scenes, enabling modular and reusable design.

State Machine: A design pattern used to control character behavior by dividing logic into different “states” such as idle, jumping, or attacking. Each state has its own script, helping to isolate logic and improve readability and control.

TileMap: A node that allows drawing levels using predefined tiles. It simplifies level design, especially for 2D games, by organizing terrain or background into a grid.

Parallax Background: A visual technique where background layers move at different speeds relative to the foreground, creating a depth illusion in 2D scenes.

AnimatedSprite2D: A node in Godot used to create animations using sprite sheets. It cycles through frames to create smooth movement or visual feedback.

CollisionShape2D: A shape assigned to a physics object to define how it interacts with other elements, such as walls, enemies, or player detection areas.

Area2D: A node used to detect objects entering or exiting a specific region. Often used for triggers, pickups, and damage zones.

Shader: A script that modifies how objects are drawn on screen. Used for visual effects like glow, dissolve, scanlines, or transitions.

Callable: In Godot scope, a Callable is a reference to a function that can be passed around and executed later. It's used to create dynamic function calls, such as registering interaction behavior without hardcoding the function call itself.

Autotile: A tile map configuration that allows tiles to automatically connect to each other based on neighbors, useful for quickly drawing large terrains with proper borders.

Itch.io: An video games and related social network website mainly focused on the indie scope, it also allows creators to upload and distribute their digital products, including games, comics, books, music, and more

6. Bibliography

Decot_YT.(2025, April 30). *Godot Tutorials* [Playlist]. Youtube
<https://www.youtube.com/watch?v=ctevHwoRI24&list=PLZz2U4fx6iuRKjeh74H0mib9F8UGnT08I>

Decot_YT.(2025, April 30). *Game Development* [Playlist]. Youtube
<https://www.youtube.com/watch?v=mrVM2zehqiw&list=PLZz2U4fx6iuTtJgwwdnCRh0vCQ7aAuvuQ>

Godot Engine Documentation. (n.d.). *Godot Engine documentation*. [Documentation]
<https://docs.godotengine.org/en/stable/>

Godot Community. (n.d.). *Godot reddit community* [Online forum]. Reddit
<https://www.reddit.com/r/godot/>

Godot Engine. (n.d.). *Godot forums* [Online forum]
<https://forum.godotengine.org>

Godot Community. (n.d.). *Godot legacy forums* [Online forum]
<https://godotforums.org>

Stack Exchange Inc. (n.d.). Godot tag – Game Development Stack Exchange. [Online forum]
<https://gamedev.stackexchange.com/questions/tagged/godot>

The Godot Barn. (n.d.). Community Q&A for Godot developers. [Online forum]
<https://thegodotbarn.com/questions>

Godot Community. (n.d.) *Godot Shaders* [Website]
<https://godotshaders.com>

Zapsplat. (n.d.). Sound effects categories. [SFX Website]
<https://www.zapsplat.com/sound-effect-categories/>

GDQuest. (n.d.). Godot tutorial: Finite State Machine (FSM) design pattern. [Website].
<https://www.gdquest.com/tutorial/godot/design-patterns/finite-state-machine/>

ShaggyDev. (2023, November 28). Godot 4: Advanced state machines. [Tutorial]
<https://shaggydev.com/2023/11/28/godot-4-advanced-state-machines/>

WordReference.com LLC. (n.d.). WordReference online dictionary. [English dictionary]
<https://www.wordreference.com>

CoolText.com. (n.d.). Cool Text: Logo and graphics generator. [Website]
<https://cooltext.com/Logos>

OpenAI. (n.d.). ChatGPT. [Chat AI]
<https://chatgpt.com>

Suno AI. (n.d.). Suno: AI music generation. [Music AI]
<https://suno.ai>

7. Get the project

DecotDev. (2025, April 30). *Heaven's Hell* [Godot project] GitHub repository
<https://github.com/DecotDev/project-movement>

7.1. Get Godot

Godot Engine. (n.d.). *Download Godot Engine*. [Game Engine]
<https://godotengine.org/download/>