
UNIVERSITAT DE BARCELONA

**Treball de Fi de Grau**

**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques i Informàtica**
**Universitat de Barcelona**

---

# Desarrollo del videojuego Divine Journey

---

**Joan Jesús Maceo Fernández**

Directora: Mireia Ribera

Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 10 de juny de 2024

## Agradecimientos

Quiero agradecer a mi tutora Mireia Ribera por toda la ayuda, recomendaciones y orientación que me ha ofrecido durante todo el desarrollo del proyecto.

También quiero agradecer a todas las personas que me han ayudado a probar el videojuego para poder encontrar y  documentar los errores.

Finalmente, quiero agradecer a mi familia, amigos y pareja por darme su apoyo sin falta para poder realizar este proyecto. Muchas gracias a todos.

## Resum

Durant el grau d'Enginyeria Informàtica, s'han après diferents tecnologies i metodologies per dissenyar diferents tipus de *software*. Tot i això, no s'ha abordat un tipus de *software* d'una de les indústries més grans actualment, la indústria dels videojocs. Aquest treball neix de l'interès professional en el desenvolupament de videojocs com a opció laboral futura.

El propòsit d'aquest TFG és aprendre i adoptar els coneixements necessaris per a desenvolupar un videojoc anomenat Divine Journey. El que es busca és aplicar els coneixements apresos durant el grau d'Enginyeria Informàtica al desenvolupament de Divine Journey, per així enfocar la carrera professional a la indústria dels videojocs.

Per fer aquest desenvolupament s'ha après a utilitzar noves tecnologies com el motor gràfic Godot i el llenguatge de programació GDScript des de zero per a la correcta implementació del videojoc juntament amb les metodologies Scrum i Kanban apreses durant el grau d'Enginyeria Informàtica. Gràcies als coneixements previs en aquestes metodologies adquirits durant el grau, el procés de desenvolupament ha estat més àgil i s'han aconseguit implementar tots els objectius exceptuant-ne un, a causa de la manca de temps i els problemes que han anat sorgint.

El treball documenta totes les fases realitzades per al desenvolupament de Divine Journey. Documenta el treball previ fet per saber quines tecnologies utilitzar i les característiques que té, l'anàlisi dels requisits per als usuaris, la planificació i distribució del desenvolupament, el disseny de l'arquitectura i l'estructura a seguir, la implementació per complir els requisits i finalment els resultats, les proves finals i les conclusions, on s'expliquen els objectius complerts del projecte.

La idea del treball és implementar la primera versió de Divine Journey compatible per a plataformes de PC (principalment a Windows) per posteriorment ensenyar aquesta primera versió a un possible inversor perquè financi la continuació del desenvolupament fins a assolir la versió comercial i treure'l a la venda a la plataforma Steam.

Al final del treball s'ha aconseguit implementar aquesta primera versió del videojoc Divine Journey per al sistema operatiu Windows. Aquesta primera versió compleix pràcticament tots els requisits que es buscaven implementar, per tant el treball ha assolit els objectius esperats per al videojoc Divine Journey.

## Resumen

Durante el grado de Ingeniería Informática se han aprendido diferentes tecnologías y metodologías para diseñar diferentes tipos de *software*. No obstante, no se ha abordado un tipo de *software* de una de las industrias más grandes en la actualidad, la industria de los videojuegos. Este trabajo nace del interés profesional en el desarrollo de videojuegos como futura opción laboral.

El propósito de este TFG es aprender y adoptar los conocimientos necesarios para desarrollar un videojuego llamado Divine Journey. Lo que se busca es aplicar los conocimientos aprendidos durante el grado de Ingeniería Informática al desarrollo de Divine Journey, para así enfocar la carrera profesional en la industria de los videojuegos.

Para realizar este desarrollo se ha aprendido a utilizar nuevas tecnologías como el motor gráfico Godot y el lenguaje de programación GDScript desde cero para la correcta implementación del videojuego junto con la metodologías Scrum y Kanban aprendidas durante el grado de Ingeniería Informática. Grácias a los conocimientos previos en estas metodologías adquiridos durante el grado, el proceso de desarrollo ha sido más ágil y se han logrado implementar todos los objetivos exceptuando uno, debido a la falta de tiempo y los problemas que han ido surgiendo.

El trabajo documenta todas las fases realizadas para el desarrollo de Divine Journey. Documenta el trabajo previo realizado para saber qué tecnologías utilizar y las características que tiene, el análisis de los requisitos para los usuarios, la planificación y distribución del desarrollo, el diseño de la arquitectura y estructura a seguir, la implementación para cumplir los requisitos y finalmente los resultados, pruebas finales y las conclusiones, donde se explican los objetivos cumplidos del proyecto.

La idea del trabajo es implementar la primera versión de Divine Journey compatible para plataformas de PC (principalmente en Windows) para posteriormente enseñar esta primera versión a un posible inversor para que financie la continuación del desarrollo hasta alcanzar la versión comercial y sacarlo a la venta en la plataforma Steam.

Al final del trabajo se ha conseguido implementar esta primera versión del videojuego Divine Journey para el sistema operativo Windows. Esta primera versión cumple con prácticamente todos los requisitos que se buscaban implementar, por lo tanto el trabajo ha alcanzado los objetivos esperados para el videojuego Divine Journey.

## Abstract

During the Computer Engineering degree, different technologies and methodologies have been learned to design different types of software. However, one type of software from one of the largest industries today, the video game industry, has not been addressed. This work is motivated by professional interest in video game development as a future career option.

The purpose of this TFG is to learn and adopt the knowledge necessary to develop a video game called Divine Journey. What is sought is to apply the knowledge learned during the Computer Engineering degree to the development of Divine Journey in order to focus the professional career in the video game industry.

To carry out this development, we have learned to use new technologies such as the Godot graphics engine and the GDScript programming language from scratch for the correct implementation of the video game along with the Scrum and Kanban methodologies learned during the Computer Engineering degree. Thanks to the previous knowledge in these methodologies acquired during the degree, the development process has been more agile and all the objectives have been implemented except for one, due to lack of time and the problems that have arisen.

The project documents all the phases carried out for the development of Divine Journey. It documents the previous work carried out to know what technologies to use and the characteristics it has, the analysis of the requirements for users, the planning and distribution of the development, the design of the architecture and structure to follow, the implementation to meet the requirements and finally the final results, the tests and the conclusions, where the achieved objectives of the project are explained.

The idea of the project is to implement the first version of Divine Journey compatible for PC platforms (mainly on Windows) and then show this first version to a potential investor to finance the continuation of development until reaching the commercial version and putting it on sale in the Steam platform.

At the end of the project, the first version of the Divine Journey video game has been implemented for the Windows operating system. This first version meets practically all the requirements that were sought to be implemented, therefore the work has achieved the expected objectives for the video game Divine Journey.

# Índice

# 1. Introducción

## 1.1. Contexto y motivación

La realización de este proyecto está motivada por la integración de los conocimientos adquiridos durante el grado de ingeniería informática junto al desarrollo de un software no abordado en el plan de estudios, como es el caso de un videojuego.

Actualmente la industria de los videojuegos es una de las más rentables, populares y en constante expansión en el sector de desarrollo de software. Por lo tanto, es importante saber adaptar las metodologías aprendidas a lo largo del grado a las necesidades reales que requiere este tipo de desarrollo y entender las necesidades de los usuarios, en este caso jugadores, a la hora de jugar un determinado género de videojuegos, para poder ofrecer un producto que las satisfaga.

De estos motivos surge la idea de desarrollar un videojuego desde cero apoyándonos en los procesos y metodologías ya conocidos durante el grado, para así, tener una buena base de conocimientos en el software de esta industria y poder seguir expandiendo estos conocimientos para desarrollar un futuro laboral orientado en el desarrollo de videojuegos.

## 1.2 Objetivos

El objetivo principal de este proyecto es diseñar y desarrollar una primera versión jugable de un videojuego independiente renderizado en 2 dimensiones (2D), de género *roguelike* (en el capítulo *2. Trabajos previos*, se explicará este género) para plataformas de ordenadores, principalmente con sistema operativo Windows. Está primera versión servirá como demostración para un posible inversor, para así poder seguir desarrollando el videojuego en trabajos futuros.

En la primera versión jugable del videojuego Divine Journey, los jugadores deben de poder explorar una mazmorra compuesta de diversas salas generada de manera procedimental, combatir contra enemigos generados en cada sala y finalmente encontrar una sala final para así completar el juego.

Estas características mencionadas del videojuego han sido la base para elaborar la lista de objetivos a conseguir en el desarrollo. Los objetivos son los siguientes:

- Sistema de generación procedimental de la mazmorra. Este sistema se encargará de generar aleatoriamente la mazmorra siguiendo unos parámetros y procedimientos especificados por el desarrollador.

- Sistema de movimiento del jugador. Este sistema se encargará de gestionar los *Inputs* de los jugadores para poder desplazar al personaje controlable por la mazmorra.

- Sistema de combate del jugador. Este sistema se encargará de la lógica para que el jugador pueda dañar a los enemigos y estos a él.

- Sistema de animaciones. Este sistema se encargará de gestionar toda la lógica de las animaciones del jugador y los enemigos, para proporcionar al jugador *feedback* visual de las acciones del videojuego.

- Sistema de sonido. Este sistema se encargará de gestionar toda la lógica de los efectos de sonido del jugador y de los enemigos.

## 1.3 Estructura de la memoria

El contenido de la memoria se ha estructurado en los siguientes apartados:

- 2. Trabajo previo: Explicación de las necesidades del proyecto junto con la investigación de las tecnologías que serán utilizadas durante el desarrollo. También incluye una investigación de videojuegos del mismo ámbito que Divine Journey para referencias.

- 3. Análisis: Explicación de los requisitos del videojuego mediante historias de usuario (User Stories) y una explicación de los elementos importantes del videojuego, su plataforma de destino y los costes de su desarrollo,

- 4. Planificación: Explicación de la metodología utilizada durante el proceso de desarrollo y la distribución temporal de las *user stories.*

- 5. Diseño:  Explicación de la conceptualización del videojuego de cara a presentarlo al inversor, la arquitectura y estructura de los sistemas implementados y los patrones de diseño utilizados.

- 6. Implementación: Explicación de los algoritmos implementados para las funcionalidades del videojuego

- 7. Resultados: Explicación de las pruebas realizadas y los resultados finales obtenidos.

- 8. Conclusiones: Se exponen las conclusiones y el trabajo futuro a realizar.

- 9. Bibliografía: Se exponen las fuentes consultadas durante el desarrollo.

- 10. Anexos: Se expone el glosario y las figuras extras de la memoria.

# 2. Trabajo previo

Como se ha explicado en la introducción, el objetivo de este proyecto es desarrollar un videojuego renderizado en 2D de género *roguelike*. Para este desarrollo se necesita de dos partes esenciales, las tecnologías para desarrollar las funcionalidades y las referencias de otros videojuegos del mismo género para guiar el diseño hacia una dirección ya aceptada por los jugadores del género.

Respecto a las tecnologías, el desarrollo necesitará: Un motor gráfico para la renderización en 2D del escenario, un software para diseñar los *sprites* y animaciones y un software para componer las piezas de audio correspondientes a los efectos de sonido. Respecto a las referencias con otros videojuegos del mismo género, se investigarán algunos de los títulos más populares junto con sus características y mecánicas.

## 2.1 Tecnologías utilizadas

Existen diversos motores gráficos que permiten un renderizado de escenarios en 2D. Algunos de los motores más destacados con esta característica se encuentran Unity, Unreal Engine y Godot. Después de investigar cada motor gráfico, se decidió utilizar Godot para desarrollar el proyecto debido a su accesibilidad para desarrolladores independientes, la alta compatibilidad para videojuegos desarrollados para ordenadores y su enfoque centrado en el renderizado de escenarios 2D.

Godot es un motor gráfico de código abierto totalmente gratuito creado por Juan Linietszki y Ariel Manzur que utiliza el lenguaje GDScript, un lenguaje de programación propio y específico para Godot que comparte grandes similitudes de sintaxis y estructura con Python. Tiene su propio entorno de desarrollo (IDE, por sus siglas en inglés) integrado, por lo tanto, no será necesario el uso de ningún IDE externo.

Para el desarrollo de los *sprites* y animaciones para el videojuego, se utilizará el software Aseprite desarrollado por Igara Studios S.A. Este software no es gratuito, pero tiene un coste bajo, el cual se especificará en el apartado *3.5 Costes del proyecto.* Aseprite permite crear *sprites* y animaciones para videojuegos renderizados en 2D basándose principalmente en el estilo *pixel art.* Este estilo será explicado más adelante en el capítulo *5.1 Conceptualización del juego.*

Para el desarrollo de las pistas de audio se utilizará el software gratuito Audacity creado por Roger Dannenberg y Dominic Mazzoni. Audacity permite la grabación y edición de audio, los cuales se pueden importar al juego para implementar los efectos de sonido.

## 2.2 Videojuegos relacionados

Como se ha explicado anteriormente, el género de Divine Journey es *roguelike*. El nombre del género proviene del videojuego del año 1980 llamado Rogue. En este videojuego, los jugadores se sitúan al principio de una mazmorra plagada de enemigos que se extiende a través de diversas salas y el objetivo es recuperar un amuleto y regresar por el camino elegido. Si los jugadores mueren durante la partida, deberán volver a comenzar desde cero. Este título fijó las bases del género *roguelike* dándole las siguientes características: Contenido aleatorio (tanto la generación de la mazmorra como los enemigos y objetos que contenga), *permadeath* (Muerte permanente) si el jugador muere deberá empezar de cero, enfoque basado en un solo jugador (existen títulos considerados *roguelike* que tienen cooperativo o directamente multijugador) y, por último, énfasis centrado más en la jugabilidad que en la narrativa del videojuego (historia, trasfondo del personaje, etc).

Para tener un guía con una base sólida para el diseño de Divine Journey, se ha realizado una investigación comparativa de algunos de los títulos del género *roguelike* más populares. Para esta comparativa se han utilizado técnicas de *benchmarking* con el propósito de identificar y adoptar las características más relevantes de los videojuegos referentes en el género. Los criterios de análisis utilizados para los juegos son, el análisis del diseño del mapa o niveles y las mecánicas jugables. Los juegos que se han analizado para esta comparativa son, The Binding of Isaac y Risk of Rain 2.

El título The Binding of Isaac ha sido la principal referencia para el diseño de Divine Journey. Destaca su diseño de nivel, caracterizado por salas sencillas y compactas, de tamaño pequeño en relación con el jugador, lo que provoca que la acción esté más concentrada, proporcionando así un aspecto positivo para el juego, una experiencia de combate más intensa para el jugador.


Figura 2.1: Habitación del juego The binding of Isaac (Fuente: https://bindingofisaac.fandom.com/es/wiki/The_Binding_of_Isaac)

En la figura 2.1 se muestra un ejemplo del diseño de este tipo de salas de The Binding of Isaac.

Este concepto se implementará en el proyecto desarrollando salas con un espacio reducido para el jugador. Además, se adoptará la mecánica de transición entre salas propia de The Binding of Isaac. Las salas son escenas y cuando se cambia de habitación se carga una escena nueva y se elimina la anterior, en el juego aplica junto con un efecto de desplazamiento de pantalla. En el proyecto se aplicará esta mecánica de cambio de escenas para el cambio de salas, pero el efecto visual será distinto al del original de The Binding of Isaac.

El título Risk of Rain 2 ha ofrecido inspiraciones derivadas de su aleatoriedad para progresar hacia el siguiente nivel. El jugador para avanzar necesita encontrar un teletransportador y activarlo, este teletransportador puede estar situado en cualquier parte del mapa, incentivando así la exploración para el jugador y ofreciendo una sensación constante de curiosidad por saber la localización del teletransportador. En el proyecto también se considerará añadir este concepto, donde para pasar de nivel el jugador necesitará encontrar la habitación del jefe final y vencerlo. Por lo tanto, el proyecto implementará la aleatoriedad de la aparición de la habitación del jefe final, haciendo que pueda aparecer en cualquier parte de la mazmorra, incentivando así también la exploración de todo el mapa por parte del jugador.

De la investigación de estos dos títulos populares se han extraído dos aspectos que guiarán gran parte del diseño de Divine Journey: El diseño de salas compactas junto a la transición entre estas salas y la aleatoriedad de aparición de la habitación del jefe final.

## 2.3 Conceptos básicos de Godot

En este apartado se explicarán unos conceptos básicos de Godot para facilitar la lectura del documento.

Godot se fundamenta en el paradigma de la programación orientada a objetos (OOP, por sus siglas en inglés), por lo tanto, el desarrollo se realizará mediante el uso de clases y objetos. No obstante, se explicará el tipo de objetos en el que se centra Godot, las escenas y los nodos.

Un videojuego desarrollado en Godot consta de dos conceptos principales: los nodos y las escenas.

Los nodos son objetos nativos de Godot, cada nodo contiene sus propios métodos y propiedades y la capacidad de heredar de otros nodos. Hay diferentes tipos: nodos centrados en la gestión de las físicas del videojuego, nodos de control centrados en la gestión de las interfaces de usuario, nodos de audio para la gestión de los efectos de sonido, entre otros.

Las escenas son la unidad principal de organización, estas contienen múltiples nodos organizados de una manera jerárquica. Las escenas también pueden ser tratadas como nodos, dando la posibilidad de poder contener diferentes escenas dentro de una escena.

# 3. Análisis

En este capítulo se explicará los requisitos del videojuego Divine Journey que los jugadores deberían esperar y la organización del desarrollo de las funcionalidades explicadas mediante *user stories.*

## 3.1 Requisitos del videojuego

Los requisitos o mecánicas de Divine Journey para los jugadores son:

- Movimiento del personaje jugable mediante *Inputs*: los jugadores podrán desplazarse por la mazmorra utilizando los *Inputs* de su teclado. Los *Inputs* permitidos serán las teclas W, A, S y D y las flechas de dirección.

- Generación procedimental de la mazmorra: la mazmorra será generada mediante un algoritmo que contará con unos parámetros que el desarrollador puede modificar para así obtener diferentes resultados de disposición de las salas de la mazmorra. La modificación de estos parámetros será realizada mediante código.

- Combate contra enemigos de la mazmorra: los jugadores podrán combatir contra los enemigos utilizando los *inputs* del ratón. Al hacer clic en el botón izquierdo del ratón, el jugador podrá realizar un ataque básico mediante un sistema basado en *hurtbox* y *hitbox.* Este sistema será explicado en el apartado *6.5 Gestión de colisiones.*

- Exploración de la mazmorra: los jugadores tendrán la posibilidad de poder moverse entre las salas de la mazmorra, para así poder encontrar la sala del jefe final.

- Menú principal: los jugadores al iniciar el videojuego observarán un menú principal en el cual podrán empezar una partida o salir del videojuego.

- Interfaz de usuario: el videojuego contará con un HUD (*Heads-Up Display*) donde se mostrará a los jugadores información sobre la partida de manera visual. Esta información consta de: una barra de vida para saber los puntos de vida restantes del jugador, el tiempo transcurrido en la partida y los fotogramas por segundo a los que se está renderizando el videojuego.

- Sistema de muerte permanente (*permadeath*): si los jugadores mueren durante la partida, volverán al menú principal donde podrán volver a empezar la partida.

## 3.2 Elementos importantes del videojuego

En este apartado se explicarán los requisitos técnicos necesarios para el correcto desarrollo del videojuego.

Para cumplir con los requisitos o mecánicas esperados en Divine Journey se necesita de ciertos elementos cruciales para el desarrollo:

- Mapa o mazmorra: contendrá diversas salas, cada sala contará con cuatro puertas que conectarán con otras salas, pero, no todas las puertas tendrán una sala conectada y, por lo tanto, no se le permitirá al jugador avanzar más por esa puerta. Tanto el jugador como los enemigos no podrán atravesar las paredes de las salas.

- Personaje principal jugable: el personaje que los jugadores controlarán. Contará con la posibilidad de desplazarse por el mapa, atacar a los enemigos y mostrará las animaciones correspondientes a las acciones del jugador.

- Enemigos: personajes no jugables (NPC, por sus siglas en inglés) que se centrarán en perseguir y atacar al jugador. Hasta que el jugador no derrote a todos los enemigos de una sala, no se podrá avanzar a la siguiente.

- Colisiones: necesarias para el correcto funcionamiento del videojuego. El jugador, los enemigos y las salas de la mazmorra contarán con colisiones para poder detectarse entre ellos y aplicar correctamente la lógica esperada.

- Animaciones: implementación de las animaciones de los diferentes *sprites* para un correcto *feedback* visual de las acciones que se estén realizando en el videojuego.

- *Scripts*: archivos de código programados en GDScript para implementar las diferentes funcionalidades esperadas.

- Sonido: implementación de las pistas de audio para reproducir los efectos de sonido durante el videojuego.

## 3.3 *User stories*

Para el desarrollo del proyecto se han tenido en cuenta dos roles de usuarios: el jugador y el desarrollador. El rol de jugador se asigna automáticamente a cualquier usuario que juegue al videojuego y el rol de desarrollador en el contexto de este proyecto es único y controla todos los aspectos del desarrollo para probar su correcta implementación de manera técnica.

A partir de estos roles se han pensado las diferentes *user stories* para implementar las funcionalidades y requisitos del videojuego. Hay dos tipos de *user stories*, épicas y no épicas. Las *user stories* épicas son muy voluminosas y requieren de un extenso trabajo para implementarlas. Este tipo de *user stories* deben ser descompuestas en *user stories* de menor tamaño para su correcta implementación. Cada *user story* requiere también de unos criterios de aceptación para poder evaluar si se ha implementado de manera correcta.

A continuación, se mostrará las *user stories* que se han priorizado para el desarrollo de la primera versión del videojuego. En el apartado *10.2 Figuras* de los anexos se muestran todas las *user stories* pensadas.

| **Como...** | **Quiero** | **Para** | **¿Es épica?** | **Criterios de aceptación** | **N.º US** |
|---|---|---|---|---|---|
| Jugador | Poder desplazarme por el mapa | | No | El personaje principal se desplaza por el mapa acorde al *input* del jugador. | US01 |
| Jugador | Ver al personaje principal dentro de los límites de la cámara | Poder visualizarlo por todo el mapa | No | Al desplazarse el personaje la cámara se desplaza con él, manteniéndolo en el centro de la pantalla. | US02 |

Figura 3.3.1: *User stories* del requisito de movimiento del jugador (Fuente: Elaboración propia)

En la figura 3.3.1 se muestran las *user stories* correspondientes a la funcionalidad para el requisito de movimiento del personaje jugable.

| **Como...** | **Quiero** | **Para** | **¿Es épica?** | **Criterios de aceptación** | **N.º US** |
|---|---|---|---|---|---|
| Jugador | Interactuar con los botones del menú principal | | Sí | | US03 |
| Jugador | Interactuar con el botón *play* del menú principal | Para poder iniciar la partida | No | Al hacer clic en el botón se carga la escena principal del juego. | US05 |
| Jugador | Interactuar con el botón *quit* del menú principal | Para poder salir y cerrar del juego | No | Al hacer clic en el botón, se cierra la ventana correspondiente al videojuego. | US06 |
| Jugador | Interactuar con el botón *settings* del menú principal | Para poder cambiar la configuración de algunos aspectos del juego | Sí | | US07 |

Figura 3.3.2: *User stories* relacionadas con el menú principal (Fuente: Elaboración propia)

En la figura 3.3.2 se muestran las *user stories* relacionadas con el menú principal. La US03 es épica y por lo tanto no tiene criterios de aceptación, debido a que se requiere dividirlas en otras *user stories* más pequeñas. Las US05, US06 y US07 componen esta división, pero la US07 resulta ser otra *user story* épica y se requeriría dividirla en otras de menor tamaño. En el proyecto no se priorizará la

implementación de la US07 debido a que se busca una primera versión jugable y, por lo tanto, se centrará más en los aspectos y requisitos ya mencionados.

| Como... | Quiero | Para | ¿Es épica? | Criterios de aceptación | N.º US |
|---|---|---|---|---|---|
| Jugador | Visualizar la barra de vida del personaje | | No | La barra de vida se muestra en la esquina inferior izquierda de la pantalla. Al recibir daño la barra se actualiza con la nueva vida del jugador. | US04 |
| Jugador | Visualizar el tiempo transcurrido en la partida | | No | El tiempo se muestra en la parte superior central de la pantalla. | US14 |
| Jugador | Visualizar el número de fotogramas por segundo de la partida | Saber la fluidez de los fotogramas de la partida | No | EL número de fotogramas por segundo se muestra en la parte superior izquierda de la pantalla. | US17 |

Figura 3.3.3: *User stories* relacionadas con el HUD (Fuente: Elaboración propia)

En la figura 3.3.3 se muestran las *user stories* correspondientes a la interfaz del jugador.

| Como... | Quiero | Para | ¿Es épica? | Criterios de aceptación | N.º US |
|---|---|---|---|---|---|
| Jugador | Realizar combos básicos con las diferentes armas | Poder dañar a los enemigos de diferentes maneras | Sí | | US18 |
| Jugador | Realizar un ataque básico | Poder dañar a los enemigos | No | Al hacer clic al botón izquierdo del ratón, el personaje principal realiza la animación de atacar. Si daña a un enemigo, la cantidad de daño se muestro visualmente con el número de daño infligido. | US08 |
| Desarrollador | Que aparezcan enemigos en las salas | Que el jugador pueda combatir contra ellos | No | Después de que transcurra un tiempo específico (tiempo de aparición o *spawn)*, aparecen un número determinado de enemigos. | US15 |

Figura 3.3.4: *User stories* relacionadas con el sistema de combate (Fuente: Elaboración propia)

En la figura 3.3.4 se muestran las *user stories* correspondientes al sistema de combate. Como se observa, la US18 es épica, por lo tanto, se requiere de su división en *user stories* más pequeñas. El desarrollo no se centrará en implementar la US18 debido a que, para la primera versión del juego para el inversor, solo es necesario la US08 para infligir daño a los enemigos.

| Como... | Quiero | Para | ¿Es épica? | Criterios de aceptación | N.º US |
|---|---|---|---|---|---|
| Desarrollador | Generar una primera versión de la mazmorra de manera procedimental | Probar la correcta generación de la mazmorra | No | Solo puede generarse una sala que contenga el jefe final. Ninguna sala puede estar conectada con más de 3 salas distintas. | US13 |

Figura 3.3.5: *User story* relacionada con la generación de la mazmorra (Fuente: Elaboración propia)

En la figura 3.3.5 se muestra la *user story* correspondiente para generar la mazmorra de manera procedural.

A continuación, se mostrarán el resto de *user story* que no fueron consideradas para el desarrollo de la primera versión de Divine Journey pero están pensadas para un futuro desarrollo. Como las siguientes *user stories* no han tenido prioridad, tampoco han sido pensados sus criterios de aceptación.

| Como... | Quiero | Para | ¿Es épica? | Criterios de aceptación | N.º US |
|---|---|---|---|---|---|
| Jugador | Realizar un desplazamiento rápido | Poder esquivar ataques enemigos | No | | US09 |
| Jugador | Utilizar diferente tipo de habilidades | Realizar diferentes tipos de ataques contra los enemigos | Sí | | US10 |
| Jugador | Poder obtener diferentes tipos de preciadores | Mejorar las estadísticas del personaje principal | Sí | | US11 |
| Jugador | Poder visualizar las estadísticas del personaje principal | | No | | US12 |
| Jugador | Poder visualizar un mapa de la mazmorra | Saber la posición del personaje principal dentro de la mazmorra | No | | US16 |

Figura 3.3.6: *User stories* no prioritarias para el desarrollo de la primera versión (Fuente: Elaboración propia)

## 3.4 Plataforma de destino

Se ha decidido que la plataforma de destino para el proyecto sea PC (Personal Computer), específicamente con el sistema operativo Windows. Para la toma de esta decisión se ha realizado una comparativa entre plataformas para elegir la más adecuada para las necesidades del proyecto.

Después de analizar la comparativa realizada para el proyecto, el sistema operativo Windows es el seleccionado para el videojuego. Los motivos de esta decisión son principalmente la amplia base de usuarios que posee Windows y la experiencia por parte del desarrollador en este sistema operativo, permitiendo una mayor agilidad a la hora de solucionar problemas que surjan durante el desarrollo del videojuego.

## 3.5 Costes del proyecto

En este apartado se explica todo el software va a ser utilizado durante el desarrollo y su coste. En el caso de utilizar un programa gratuito, el coste quedará marcado como 0 euros.

| **Software** | **Coste** | **Url** |
|---|---|---|
| Motor gráfico Godot | 0 euros | https://godotengine.org/ |
| Programa Aseprite para el desarrollo de los sprites del juego | 18 euros | https://aseprite.org/ |
| Programa Audacity para el desarrollo del audio del juego | 0 euros | https://www.audacityteam.org/ |

Figura 3.5.1: Coste del software para el desarrollo (Fuente: Elaboración propia)

Para la parte del coste humano, es decir, la parte de desarrollo y mantenimiento del videojuego, el coste de un trabajador Junior es de 8 euros la hora, pero hay que tener en cuenta que este tipo de desarrollo requeriría de más trabajadores, por lo tanto, estimaremos el coste en unos 12 euros la hora.

El tiempo de trabajo se ha distribuido en un total de 17 semanas (empezó casi a la par con el segundo semestre, 12 de febrero) de las cuales 4 se dedicarán a la memoria y no contarán. Con un trabajo diario de 6 horas durante 3.75 días a la semana durante 13 semanas, el coste humano para el desarrollo es de 3510 euros. Esto se añadirá al coste del software.

| Tipo | Coste |
|---|---|
| Motor gráfico Godot | 0 euros |
| Programa Aseprite para el desarrollo de los sprites del juego | 18 euros |
| Programa Audacity para el desarrollo del audio del juego | 0 euros |
| Coste humano | 3510 euros |
| Coste total desarrollo | 3528 euros |

Figura 3.5.2: Coste total del desarrollo de Divine Journey (Fuente: Elaboración propia)

# 4. Planificación

En este capítulo se mostrará la metodología utilizada para el desarrollo y la distribución del trabajo y de las *user stories.*

## 4.1 Metodologías utilizadas

Las metodologías que se aplicarán en el proyecto serán Scrum y Kanban, principalmente se utilizará Scrum.

Scrum facilita la organización y gestión del trabajo en equipos de desarrollo. Su funcionamiento se basa en *sprints*, períodos fijos de tiempo donde se completan las tareas designadas para el período correspondiente. Estos *sprints* generalmente tienen una duración entre 2 y 4 semanas. Las ventajas que aporta Scrum son: transparencia del progreso del trabajo a través de revisiones periódicas del desarrollo, adaptabilidad a los cambios y problemas mediante ajustes rápidos en el desarrollo, mejora continua del flujo de trabajo y técnica de desarrollo mediante retrospectivas y reflexiones al final de cada sprint.

Kanban se centra principalmente en la gestión del flujo de trabajo de manera visual mediante un tablero. Cada tarea de desarrollo se representa en una tarjeta y se va desplazando entre columnas, las cuales indican el estado de cada tarea. Estos estados son principalmente, *To Do*, *Doing* y *Done*. Las ventajas que aporta Kanban son: visualización clara del progreso y estado de cada tarea, flexibilidad permitiendo cambios y ajustes cuando sean necesarios y reducción del trabajo en proceso (WIP, por

sus siglas en inglés), permitiendo la identificación visual de cuellos de botella y manteniendo así el flujo de trabajo.

Con estas dos metodologías se ha organizado el flujo y distribución del trabajo para el correcto desarrollo del videojuego Divine Journey.

## 4.2 Organización de Scrum y Kanban en el desarrollo

En el contexto del proyecto, que es desarrollado por una sola persona, el *Scrum Team* se compondrá únicamente del desarrollador, quien asumirá también el rol de *Product Owner* (encargado de compartir la visión del producto) debido a su control de la visión del proyecto. En el proyecto, el rol de *Scrum Master* (encargado de dirigir al *Scrum Team*) se ha considero innecesario y, por lo tanto, no será considerado para el desarrollo. Además, debido a que el *Scrum Team* está compuesto por una sola persona, la práctica de reuniones diarias *(daily meeting*) de unos 5 minutos no se tendrá en cuenta y no se realizará.

Respecto a la organización de Scrum, los s*prints* para el proyecto han sido establecidos para durar un periodo de dos semanas. Este tamaño reducido de tiempo ha sido considerado adecuado para el proyecto dada la falta de experiencia por parte del desarrollador. De esta manera, se podrá iterar de manera más rápida, permitiendo así un aprendizaje y adaptación continua a las tecnologías utilizadas en el proyecto, principalmente el motor gráfico Godot.

La división de las *user storie*s se convertirán en *sprint tasks* y se estimará su duración en horas para el desarrollo en el apartado *4.3 Estimación y distribución de las user stories.*

Para la organización de Kanban, la visualización del trabajo se realizará a través un tablero Kanban en el cual se mostrará todo el flujo de trabajo, el tablero estará dividido en 4 columnas.

La primera columna será el *Backlog* donde se colocarán las tareas pendientes de desarrollar, la siguiente columna será la columna *ready* donde se colocarán las tareas que estén listas para iniciar su desarrollo, la siguiente columna será *in progress* donde se colocarán las tareas que se estén actualmente en desarrollo y por último la columna “Done” donde se colocarán las tareas completadas y, por lo tanto, hayan terminado su desarrollo.

Para limitar el flujo de trabajo y evitar así cuellos de botellas mencionados anteriormente, cada columna contará con un límite de tareas que pueden contener. Con esta implementación se busca reducir la sobrecarga de trabajo durante el desarrollo del juego.

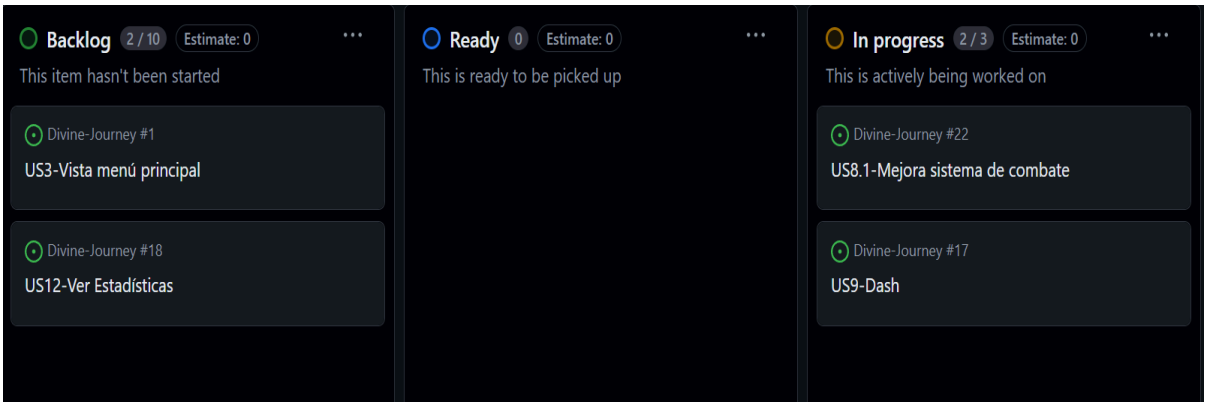


Figura 4.2.1: Kanban de GitHub del proyecto Divine Journey (Fuente: Elaboración propia)

En la figura 4.2.1 se pueden observar las columnas *Backlog* y *In Progress*. Estas columnas tienen una limitación de elementos que pueden contener, *Backlog* puede contener hasta máximo 10 y *In Progress* hasta máximo 3.

## 4.3 Estimación y distribución de las *user stories*

Para poder distribuir las *user stories* en *sprints* se ha realizado una estimación en horas de su duración y se han ordenado por prioridad.

| ***User Story*** | **Descripción** | **Estimación en horas** |
|---|---|---|
| US01 | Desplazar por el mapa al personaje principal | 3 |
| US02 | Personaje principal dentro de los límites de la cámara | 2 |
| US05 | Interactuar con el botón *play* del menú principal | 2 |
| US06 | Interactuar con el botón *quit* del menú principal | 2 |
| US08 | Realizar ataque básico | 10 |
| US04 | Visualizar la barra de vida | 5 |
| US13 | Generación procedimental de la mazmorra | 30 |
| US15 | Aparición de enemigos en las salas de la mazmorra | 20 |
| US07 | Visualizar los fotogramas por segundo de la partida | 1 |
| US04 | Visualizar el tiempo transcurrido en la partida | 1 |

Figura 4.3.1: Tabla de prioridad y estimación en horas de las *user stories* (Fuente: Elaboración propia)

En la figura 4.3.1 se muestra la prioridad final de las *user stories*, para implementar en el desarrollo de la primera versión de Divine Journey. Las *user stories* están ordenadas en orden descendente de prioridad, es decir, la primera es la que tiene mayor prioridad.

La distribución del proyecto es de unas 13 semanas, por lo tanto, se ha decidido distribuir el desarrollo a lo largo de 5 *sprints*. Cada *sprint* tendrá una duración de dos semanas como ya se ha mencionado en el apartado *4.2 Organización de Scrum y Kanban en el desarrollo de Divine Journey*. El tiempo extra será utilizado como margen para solucionar los posibles errores que surjan durante el desarrollo, realizar las pruebas del funcionamiento e investigar documentación e información para ayudar a la implementación. La planificación original para distribuir las *user stories* en los *sprints* es la siguiente:


Figura 4.3.2: Diagrama de Gantt de la distribución de las *user stories* (Fuente: Elaboración propia)

La razón de esta distribución viene dada de los objetivos de desarrollo que se querían cumplir en cada *sprint.* Los objetivos que se buscaban en cada *sprint* son:

- Sprint 1: En este *sprint* se buscaba implementar las bases del videojuego para poder desarrollar sobre ella los siguientes *sprints*. Se buscaba implementar las funcionalidades más básicas, el movimiento del jugador, la cámara y los botones *play* y *quit* del menú principal para poder iniciar partida y salir del juego.

- Sprint 2: En este *sprint* se buscaba implementar las funcionalidades para el combate del videojuego. Se querían implementar las funcionalidades necesarias para que el jugador pueda realizar un ataque básico y visualizar la vida del personaje.

- Sprint 3: Con las bases ya implementadas en el videojuego en los anteriores dos *sprints*, en este *sprint* se buscaba implementar una de las funcionalidades más importantes y costosas del desarrollo, la generación procedimental de la mazmorra. Es por este motivo que en este *sprint* solo se ha distribuido una *user story*.

- Sprint 4: En este *sprint* se buscaba implementar la última parte del sistema de combate del videojuego, los enemigos. El propósito era implementar tanto los enemigos como su aparición en las salas de la mazmorra.

- Sprint 5: En este *sprint* se buscaba implementar los últimos dos elementos de la HUD del jugador y realizar las pruebas del funcionamiento del juego con otros jugadores.

La organización completa de todo el proyecto es la siguiente:

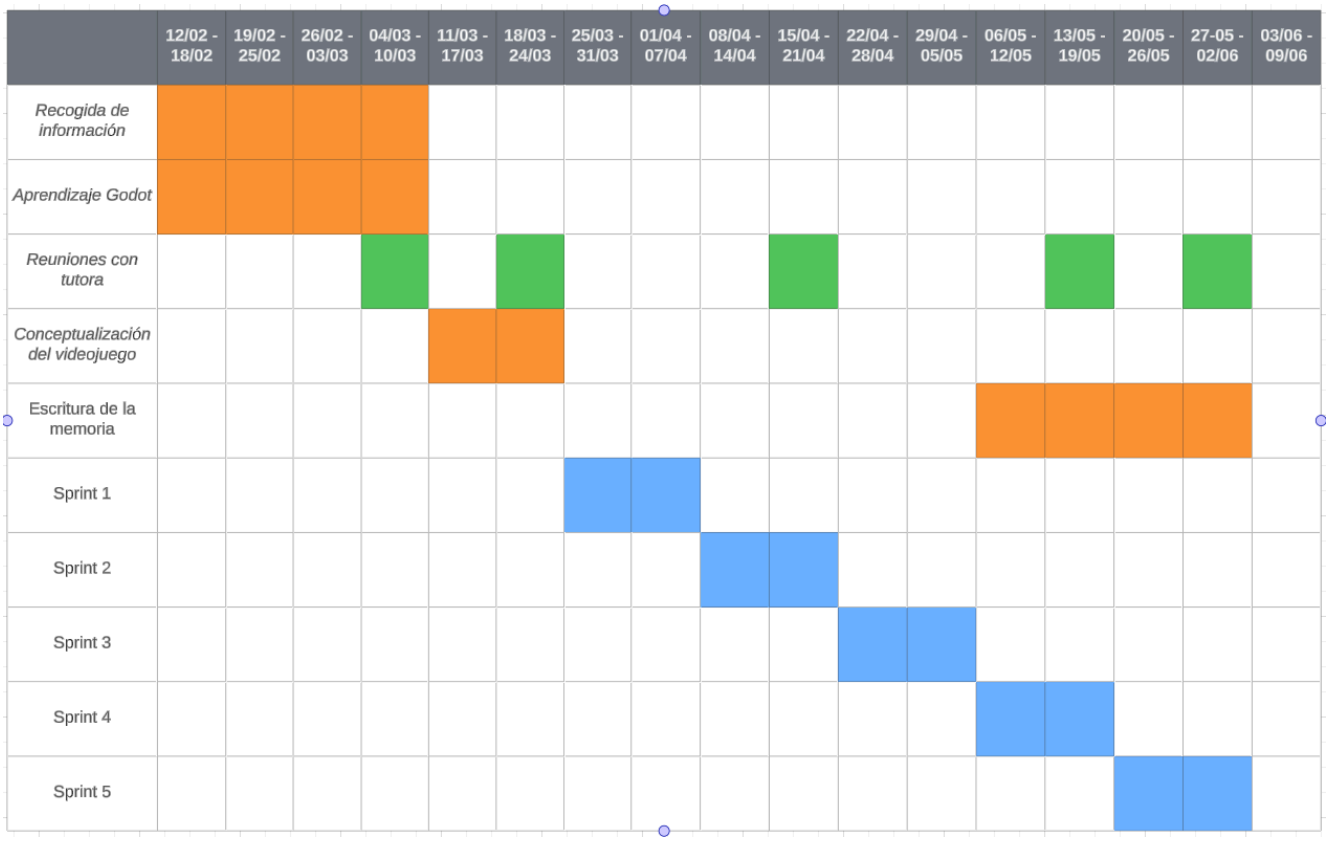


Figura 4.3.3: Diagrama de Gantt de todo el proyecto (Fuente: Elaboración propia)

### 4.4 Adaptación de la planificación

Gracias a la naturaleza de la metodología Scrum, el proyecto se fue adaptando a los diferentes cambios y problemas de implementación que surgieron durante el desarrollo. En el *sprint* 2 la *user story* resultó ser más complicada de lo pensado originalmente, este cambio provocó que no se lograron implementar ninguna de las 2 *user stories* pensadas para el *sprint* 2 y se centró en investigar la manera para implementar correctamente la US08 del ataque básico.

EL cambio en el *sprint* 2 provocó alargar las *user stories* pensadas para el *sprint* 3. Las *user stories* del *sprint* 2 fueron completadas en el *sprint* 3, pero este hecho provocó que la US13 de la generación de la mazmorra no pudiese ser completada y fue alargada al siguiente *sprint*. La US13 también resultó ser más larga y costosa de lo pensado originalmente y no solo fue alargada hasta el *sprint* 4 sino que se completó en el *sprint* 5. Este hecho provocó que las pruebas con jugadores se realizarán en la última semana en vez de en el *sprint* 5.

El resto de *user stories* fueron completadas en sus *sprints* correspondientes.

# 5. Diseño

Este capítulo explicará la estructura utilizada para el proyecto. Esto incluye la arquitectura del proyecto en Godot, la estructura de las clases, el diagrama de clases y los patrones de diseño utilizados. También se explicará la conceptualización del videojuego Divine Journey con el propósito de ser presentada a un futuro inversor.

### 5.1 Conceptualización de Divine Journey

Este apartado explicará la información referente al videojuego Divine Journey para su implementación final pensada para un desarrollo futuro.

<u>Descripción</u>

El videojuego Divine Journey se define como un *roguelike* clásico al estilo del propio Rogue donde el objetivo del jugador es encontrar la habitación del jefe final para derrotarlo. Para que aparezca el jefe y pueda ser derrotado, el jugador tendrá que encontrar una habitación clave (key room) en la cual se encontrará un objeto que permitirá que el jefe final aparezca en su sala correspondiente.

La versión final del juego incluiría un total de 4 temáticas distintas para el mapa, es decir, cuatro posibles escenarios donde se generaría la mazmorra. Las temáticas posibles son: medieval, futurista, china y japonesa.

El sistema de combate es sencillo, centrado en ataques básicos y el uso de habilidades para infligir daño en área. Además, el jugador podrá utilizar potenciadores para mejorar sus estadísticas.

Sinopsis

El personaje principal no se acuerda de por qué está en este mundo y su objetivo será enfrentarse, sin saber por qué, a diferentes oleadas de enemigos, y a un fenómeno aún más confuso, cada sala en la que está tiene una ambientación realmente particular y cada vez que consigue derrotar a los enemigos, una puerta misteriosa se abre y conduce a otra sala, pero con una ambientación completamente distinta. El personaje principal está totalmente confundido, pero por alguna razón sabe que la forma de descubrir qué está pasando y cuál es su verdadera identidad es derrotando a los monstruos y continuar avanzando por las salas. El personaje principal, aún sin saber qué está pasando exactamente, siente que sus conocimientos de combate y agilidad de movimiento tienen un nivel de alguien totalmente acostumbrado. ¿Qué misterio explicará todos estos sucesos?

Ambientación

Como ha sido introducido anteriormente en la Descripción del juego, habrá un total de 4 temáticas o ambientaciones posibles para el mapa, medieval, futurista, china y japonesa, cada una de estas ambientaciones tendrá enemigos diferentes y potenciadores diferentes.

La ambientación medieval tiene como referencia los antiguos patios de armas de esta época. Como referencia se ha utilizado la siguiente localización.

La ambientación futurista tiene como referencia ciudades del estilo “cyberpunk", las cuales representan un futuro en decadencia. Como referencia se ha utilizado la siguiente imagen.

La ambientación china tiene como referencia localizaciones famosas del país de China. Algunas de estas localizaciones son: la Ciudad prohibida, Beijing Opera House y el desierto de Gobi

La ambientación japonesa tiene como referencia, al igual que con China, localizaciones famosas del país de Japón. Algunas de estas localizaciones son: Dotonbori, Nara y Kyoto.

Personaje principal

El personaje principal tiene una apariencia sencilla, con el pelo rizado, camiseta azul y pantalones de color amarillo. Es capaz de utilizar diferentes tipos de armas incluyendo sus puños, espadas, lanzas y hachas. Esta variedad de armas viene dada por la capacidad de combate elevada del personaje principal, permitiéndole dominar un gran abanico de armas. También es capaz de utilizar diferente tipo de habilidades para derrotar a los diferentes enemigos que se encuentre durante su exploración por la mazmorra.

Enemigos

Los enemigos más abundantes del juego son los conocidos duendes (goblins). Este tipo de enemigo es un recurso frecuentemente utilizado en gran cantidad de videojuegos. Destacan por su tono de piel verdoso y sus orejas puntiagudas, además, son conocidos por su agresividad hacia cualquiera que no sea de su especie. Como jefe final, Divine Journey contará con un duende más poderoso.

Estilo artístico

El estilo artístico escogido para el videojuego es el llamado *pixel art*. Este estilo era principalmente utilizado en los videojuegos más antiguos debido a las limitaciones del hardware de la época. Hoy en día ha evolucionado hasta el punto de convertirse en todo un estilo propio.

Este estilo se caracteriza por un enfoque centrado en minimizar los detalles de los diseños manteniendo la esencia y estructura de éste. Un ejemplo de un escenario en pixel art es el siguiente:

Figura 5.1.1: Captura del videojuego Eldest Soul de su escenario en *pixel art* (Fuente: https://www.eurogamer.es/eldest-souls-se-retrasa-hasta-otono)

## 5.2 Arquitectura de Divine Journey

La arquitectura que sigue el videojuego Divine Journey es el patrón ECS (Entity-Component-System). Las entidades corresponden en el proyecto a los identificadores del jugador y enemigos que se utilizarán para acceder a los datos correspondientes. Los componentes son los datos de las entidades, es decir, las propiedades de estas entidades. En el caso del proyecto, estos datos son, la vida de los jugadores y enemigos, su posición, orientación, etc. En los componentes no se incluye la lógica del

videojuego. El sistema es donde reside la lógica del videojuego que procesa los datos de los componentes y el comportamiento de estos en el videojuego.

Para la implementación de las funcionalidades del videojuego, se necesitarán diferentes elementos: Los *inputs* entrados por el jugador serán recogidos en variables para su utilización en las funcionalidades de movimiento y ataque del personaje principal.

Los *scripts* contendrán las propiedades de las diferentes entidades y los métodos encargados de la lógica del videojuego. Los *scripts* pueden agregarse a los nodos en Godot para poder acceder a las propiedades y métodos desde el nodo.

Para ofrecer al jugador un *feedback* visual cada vez que se realice una acción o funcionalidad en el videojuego como un ataque del jugador o de un enemigo, es importante la gestión correcta de las animaciones. Esta gestión será explicada en detalle en el apartado *6.4 Gestión de las animaciones*.

## 5.3 Estructura de las clases

En este apartado se explicarán las estructuras de las clases (scripts) más importantes.

Script del jugador

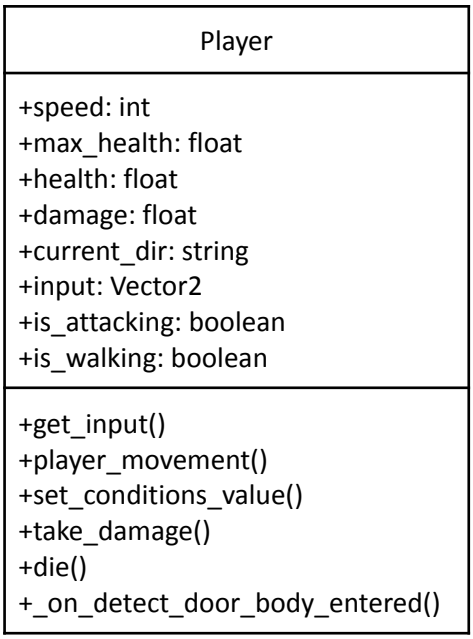


Figura 5.3.1: Propiedades y métodos de la clase Player (Fuente: Elaboración propia)

La clase Player contiene: Speed que es la velocidad a la que se desplaza el jugador, max_health es la vida máxima, health es la vida actual, damage es el daño que puede infligir. Current_dir es la dirección en la que está orientada expresada en un string para poder reproducir la animación correcta correspondiente a la dirección del jugador. Input es la entrada por teclado del jugador, esta

información se guarda en un vector de dos dimensiones. Is_attacking representa mediante un booleano si el jugador está atacando o no, de esta manera se controla la reproducción de las animaciones de atacar. Is_walking representa mediante un booleano si el jugador está moviéndose o no, de esta manera se controla si reproducir las animaciones de caminar o las animaciones de estar quieto.

Get_input() controla la lógica para recoger el *input* por teclado del jugador y almacenarlo en la variable input. Player_movement() gestiona la lógica del movimiento del jugador teniendo en cuenta el valor de la variable input. Set_conditions_value() gestiona los parámetros que controlan la reproducción de las animaciones, en el apartado *6.4 Gestión de animaciones* se verá en más detalle esta implementación. Take_damage() gestiona la lógica para disminuir los puntos de vida del jugador al recibir daño de un enemigo. Die() gestiona la lógica de fin de partida cuando los puntos de vida del jugador llegan a 0. Finalmente, _on_detect_door_body_entered() gestiona la lógica para cambiar de sala en la mazmorra.

Script del enemigo

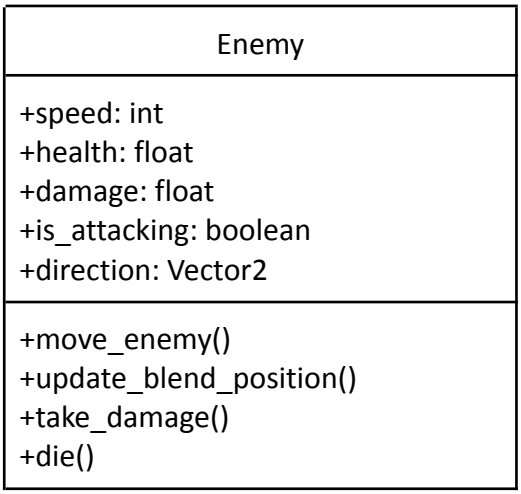


Figura 5.3.2: Propiedades y métodos de la clase Enemy (Fuente: Elaboración propia)

La clase Enemy contiene varias similitudes con la clase Player, las propiedades speed, health, damage e is_attacking tienen la misma funcionalidad que en la clase Player. Direction es el equivalente a la propiedad input de la clase Player, con la diferencia de que direction dependerá de la dirección a la que se tenga que desplazar el enemigo para perseguir al jugador. Move_enemy() gestiona la lógica para el movimiento del enemigo y perseguir al jugador. Update_blend_position() gestiona la misma lógica para las animaciones que el método Set_conditions_value() de la clase Player. Take_damage() gestiona la misma lógica para recibir daño que en la clase Player. Die() gestiona la lógica para eliminar a los enemigos de la escena una vez sus puntos de vida lleguen a 0

Script de las salas

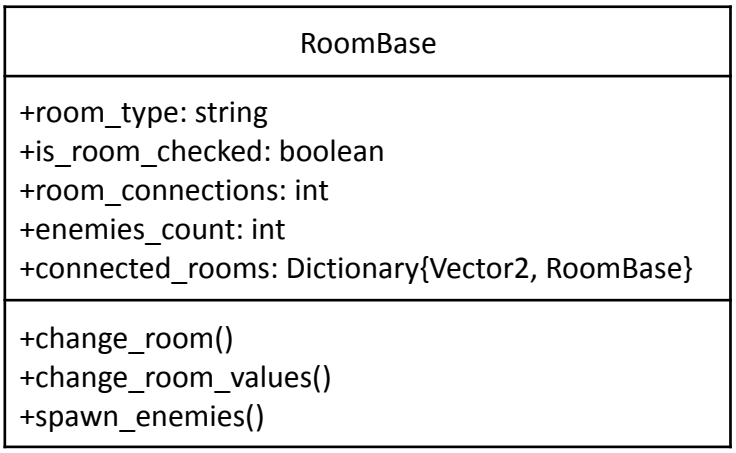


Figura 5.3.4: Propiedades y métodos de la clase RoomBase (Fuente: Elaboración propia)

La clase RoomBase contiene: Room_type que representa el tipo de la sala con un *string*. Is_room_checked es un booleano utilizado para el algoritmo de generación de la mazmorra para saber si ya se ha revisado esa sala. Room_connections es un representa el número total de salas conectadas. Enemies_count representa el número total de enemigos de la sala. Connected_rooms es un diccionario que como clave tiene un vector de dos dimensiones que representa una de las cuatro posibles conexiones y como valor una referencia a la sala conectada en esa dirección (si no hay ninguna sala conectada, el valor es *null*). Change_room() gestiona la lógica para cambiar las salas. Change_room_values() gestiona la lógica para cambiar los valores de las propiedades de la sala. Spawn_enemies() gestiona la lógica para generar enemigos en el punto de aparición de la sala.

Estas serían las clases más importantes en el desarrollo del videojuego.

## 5.4 Diagrama de escenas

En este apartado se mostrará el diagrama de las diferentes escenas diseñadas para el videojuego y la relación entre ellas.

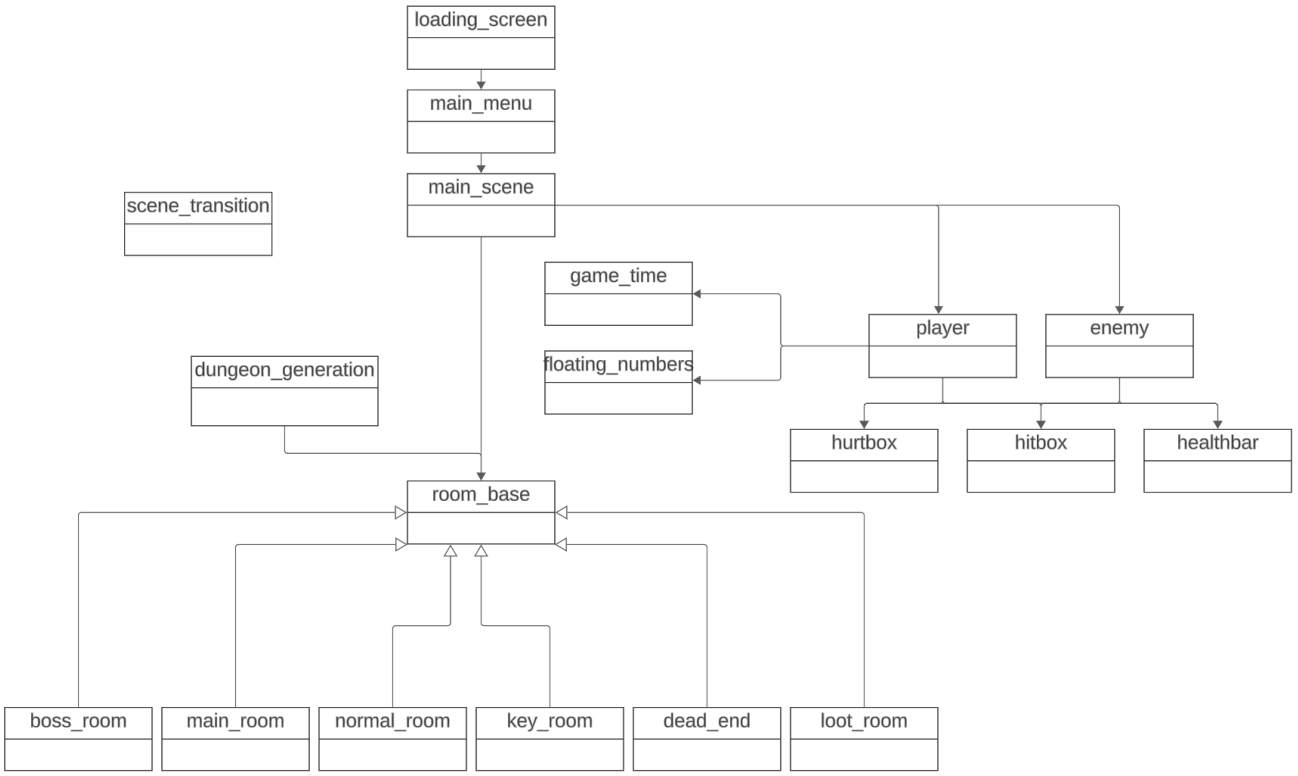


Figura 5.4.1: Diagrama de escenas del videojuego Divine Journey (Fuente: Elaboración propia)

La figura 5.4.1 muestra con una flecha con la punta negra las escenas que contiene otras escenas o al acabar su ejecución pasan a otra escena. Para mostrar la herencia de las escenas de cada sala de la escena room_base se utiliza una flecha con la punta blanca. Con este diagrama se entiende la dependencia que tienen entre sí las escenas.

## 5.5 Patrones de diseño

Como se ha explicado en el apartado *5.2 arquitectura del videojuego*, uno de los patrones utilizados es el ECS (Entity-Component-System). Otro patrón utilizado principalmente en el proyecto es el Singleton.

El patrón Singleton es un patrón de diseño que consiste en crear una instancia única de una clase o escena en el caso del proyecto con Godot y proporciona un punto de acceso global a esta instancia. Una de las mecánicas básicas del videojuego es el cambio de sala para poder seguir explorando la mazmorra. Esta funcionalidad está implementada de manera que cuando el jugador quiera cambiar de sala, se cambia la escena completamente, por lo tanto, se perderá toda la información de la sala en la que estaba el jugador (información como los enemigos que quedan). Para solucionar este problema se ha utilizado el patrón Singleton para la persistencia de datos a través de las salas cuando estas cambian.

Para el proyecto se han creado 3 instancias únicas para mantener la información de las salas y del jugador cuando se realicen los cambios de escena. En Godot para crear una instancia única se utiliza la carga automática de nodos y *scripts* conocida como *autoload*. Esto es debido a que GDScript no

soporta variables globales por diseño. Al añadir las escenas o *scripts* al *autoload* estas se cargarán automáticamente en cualquier escena.

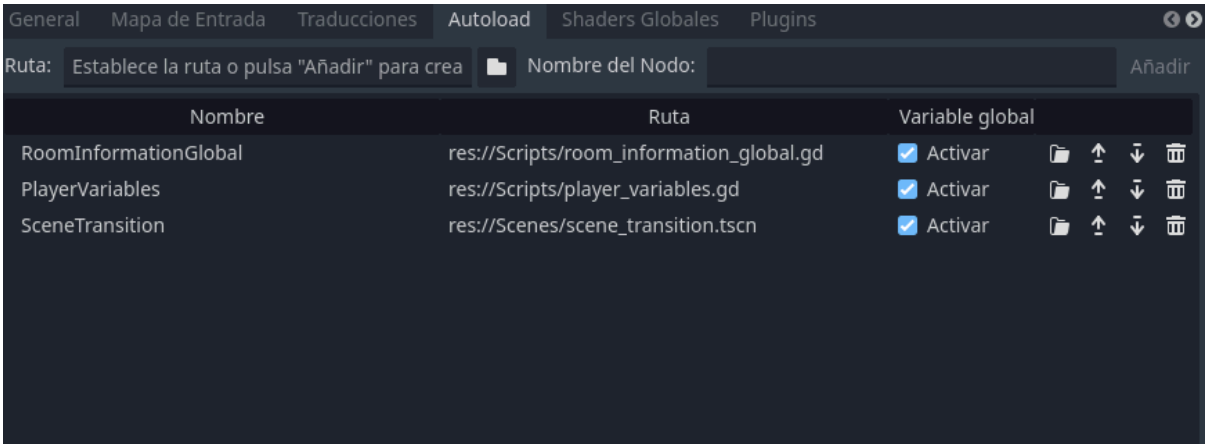


Figura 5.5.1: Captura del autoload del proyecto en Godot (Fuente: Elaboración propia)

Como se puede observar en la figura 5.5.1, el proyecto contiene tres instancias únicas, RoomInformationGlobal, PlayerVariables y SceneTransistion. A continuación, se explicará qué información guarda cada una.

RoomInformationGlobal guarda variables con las escenas cargadas de cada una de las salas, así se evita tener que cargarlas durante el juego, también contiene una variable con el *script* de la sala actual y toda la información de sus variables. Por último, se guarda una variable con la dirección correspondiente a la entrada del jugador en la sala, así al cargar la escena se elige correctamente el punto de aparición del jugador.

PlayerVariables contiene: una variable con la posición donde el jugador debe de aparecer al cargar una sala, una variable con los segundos transcurridos en la partida para que se mantengan al cargar las escenas y, para terminar, dos variables que guardan la vida actual y máxima del jugador para que se mantenga entre escenas.

Por último, SceneTransition simplemente es una escena que sirve para crear un efecto de *fade* (esfumarse) cuando se cambia de una sala a otra.

# 6. Implementación

En este capítulo se explicarán las diferentes escenas y algoritmos implementados para conseguir el correcto funcionamiento de las características del videojuego mencionadas en el apartado *3.1 Requisitos del videojuego*.

## 6.1 Escena del jugador

En este apartado se explicará el desarrollo de la escena Player y los elementos más importantes que componen esta escena.

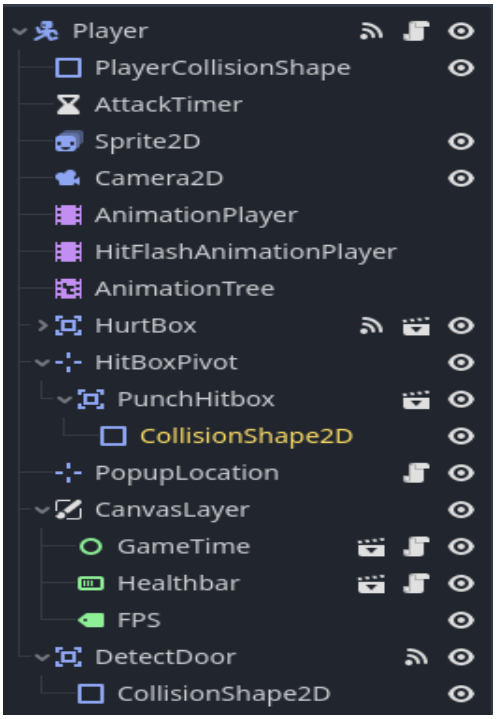


Figura 6.1.1: Captura de la estructura de los nodos del jugador (Fuente: Elaboración propia)

Es importante proporcionar al jugador una manera de interactuar y colisionar con el entorno del mapa. De esta manera el jugador podrá reaccionar a colisiones y eventos relacionados, proporcionando así una correcta inmersión en el videojuego.

Para empezar, el nodo raíz de la escena es un nodo del tipo CharacterBody2D, este nodo está pensado para ser utilizado en objetos físicos y proporciona una API de alto nivel para poder movernos detectando colisiones. Este tipo de nodo está pensado para ser controlado por el usuario o jugador recibiendo su Input. Para gestionar las colisiones con las paredes de las salas se ha combinado este nodo CharacterBody2D con un nodo CollisionShape2D. Este último permite definir una forma de colisión para el jugador con la cual interactuará con las paredes de las salas. Para el proyecto, la forma para la colisión es un rectángulo, ya que encaja muy bien con el cuerpo del personaje. En el apartado

*6.4 Gestión de animaciones* se explicará con más detalle la gestión de las colisiones que ha sido diseñada para el proyecto.

Para todo el apartado de animaciones del jugador se ha utilizado una combinación de tres nodos llamados, AnimatedSprite2D, AnimationPlayer y AnimationTree. En el AnimatedSprite2D se han juntado los Sprite para formar las animaciones.

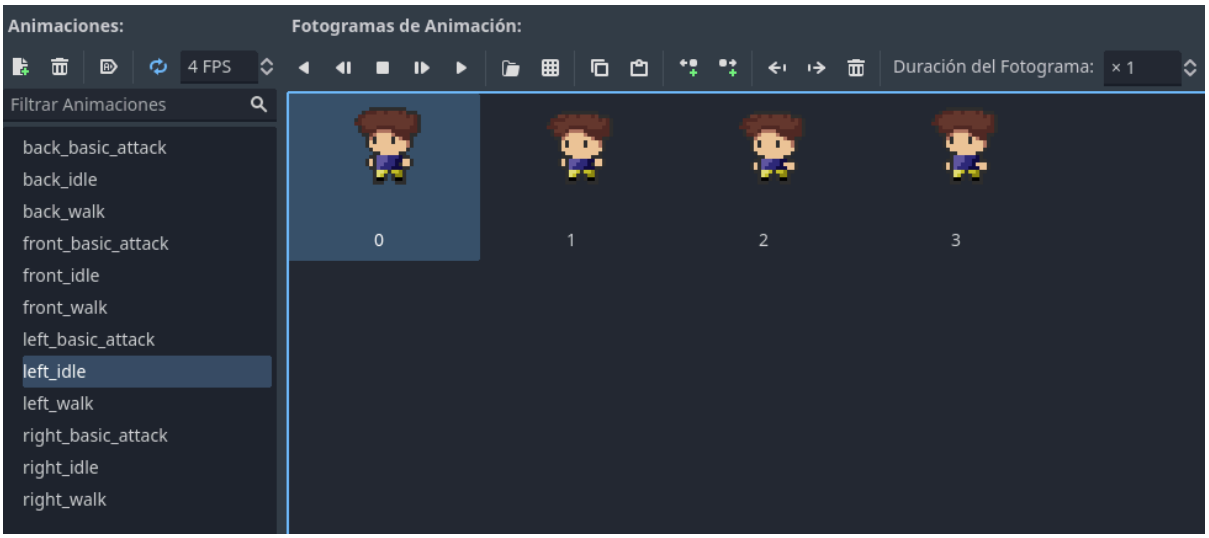


Figura 6.1.2: Captura de la pestaña SpriteFrames del jugador (Fuente: Elaboración propia)

En la figura 6.1.2 puede observarse la pestaña SpriteFrames del editor de Godot donde se han creado las distintas animaciones añadiendo los diferentes *sprites*. En la parte izquierda de la imagen están los nombres de todas las distintas animaciones que han sido creadas para el jugador y en la parte derecha están los diferentes fotogramas que componen la animación que esté seleccionada.

Para la funcionalidad de mantener al personaje principal dentro de los límites de la pantalla, la escena del jugador contiene un nodo tipo Camera2D (es la única escena que contiene un nodo de cámara) que se encargará de mantenerlo dentro de los límites de la pantalla.

Para gestionar el sistema de combate, que consiste en recibir daño de los enemigos e infligir daño a estos, se han utilizado dos escenas que se explicarán en detalle más adelante en el apartado *6.5 Gestión de colisiones*.

El nodo AnimationPlayer llamado HitFlashAnimationPlayer se encarga de crear un efecto de “flash” en el jugador volviéndolo blanco durante medio segundo cuando recibe daño. La escena Healthbar representa la barra de vida del jugador y reacciona a los cambios que presente la salud del personaje utilizando las señales de Godot.

Para gestionar el cambio de salas para explorar la mazmorra, se utiliza un nodo tipo Area2D llamado DetectDoor. Este nodo al igual que el CharacterBody2D viene con un CollisionShape2D que permite definir una forma de colisión. El nodo Area2D permite detectar cuándo un objeto de colisión entra o

sale de su área, está funcionalidad se utiliza para detectar las puertas de las salas y así cambiar de escenas en la mazmorra.

Para finalizar, se utiliza un nodo tipo CanvasLayer. Este nodo permite añadir por separado una capa extra de renderizado 2D. El *viewport* de Godot, es la capa de renderizado principal que permite mostrar todos los nodos en la vista del juego. Si un CanvasLayer aparte es añadido, esto permite, por ejemplo, tener elementos de UI (User Interface) fijos en la pantalla sin ser afectados por los movimientos de la cámara.

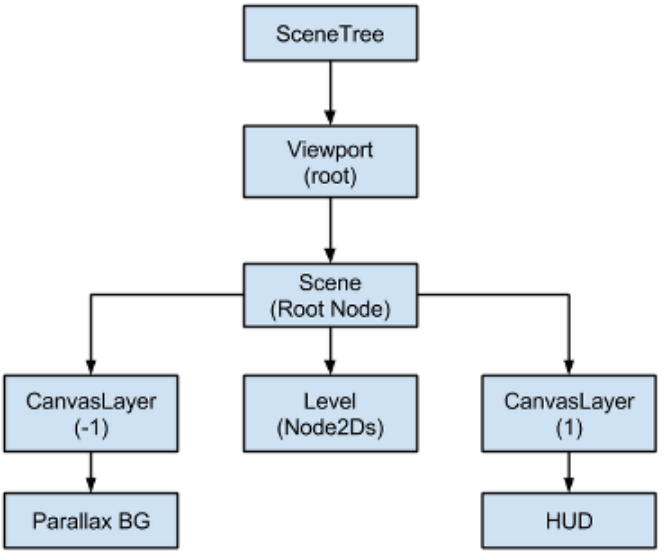


Figura 6.1.3: Diagrama de Viewport con CanvasLayer (Fuente: https://docs.godotengine.org/en/stable/tutorials/2d/canvas_layers.html)

En la figura 6.1.3 se muestra la estructura que quedaría de los Canvas del ejemplo explicado anteriormente. En el caso del proyecto, se utiliza un CanvasLayer para mostrar de manera fija tres elementos UI: la escena GameTime que es una escena que nos muestra en la parte central superior de la pantalla el tiempo transcurrido en la partida, la escena Healthbar que representa la salud actual del jugador y, por último, un nodo de tipo Label (básicamente permite colocar un texto en la escena) que muestra las FPS actuales del juego.

## 6.2 Escena de los enemigos

La escena Enemy es una escena que sirve para definir a los enemigos. Esta escena contiene al enemigo conocido como Goblin (duende) y es el que se ha usado para crear y probar las funcionalidades relacionadas con los enemigos.

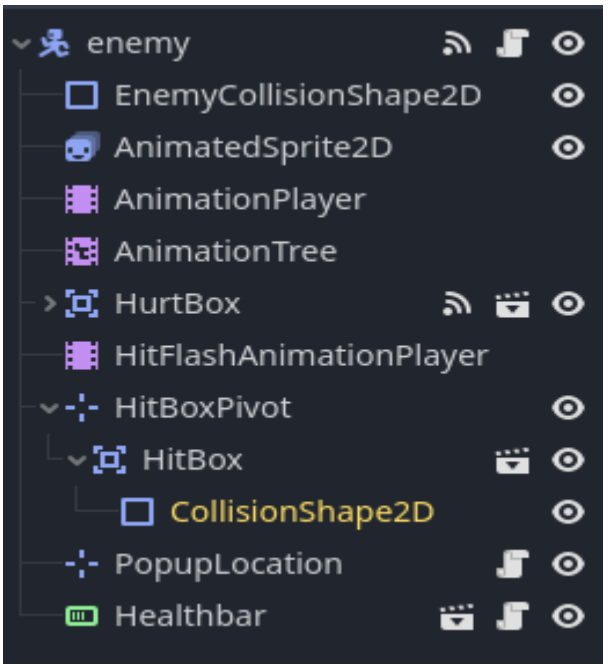


Figura 6.2.1: Captura de la estructura de los nodos del enemigo (Fuente: Elaboración propia)

El nodo raíz de la escena es un nodo del tipo CharacterBody2D, al igual que en el caso de la escena del jugador, pero con la diferencia de que este CharacterBody2D no recibirá ningún Input por parte del jugador y en cambio se moverá constantemente persiguiendo al jugador por la habitación correspondiente. Esto lo hará con una codificación de movimiento básica que consiste en dos puntos, la posición del enemigo y la posición del jugador, con estos dos puntos se forma un vector del enemigo al jugador, el vector es normalizado (dividir por su longitud) y se obtiene la dirección que tiene que seguir el enemigo para el jugador, por lo tanto, este vector será el input que recibirá el enemigo para moverse. Al ser las salas de la mazmorra cuadradas y abiertas sin objetos con colisión por delante, este tipo de movimiento es simple y se adapta muy bien a este diseño de salas.

Al igual que la escena del jugador, también contiene un nodo de tipo CollisionShape2D para las colisiones al cual se le ha asignado la forma de un rectángulo, la cual encaja correctamente con el cuerpo de los enemigos. El resto de los nodos que contiene la escena se han visto anteriormente en el apartado *6.1 Escena del jugador*. También contiene: los nodos AnimatedSprite2D, AnimationPlayer y AnimationTree para la gestión de las animaciones del enemigo, las escenas HitBox y HurtBox para gestionar el sistema de combate y la escena HealthBar que representa la vida del enemigo.


Figura 6.2.2: Captura de la pestaña SpriteFrames del enemigo (Fuente: Elaboración propia)

La lógica que siguen los enemigos para atacar al jugador es la siguiente. Primero, persiguen al jugador por toda la habitación, cuando el jugador se encuentra en rango de ataque (esto se controla con una variable en el código) entonces el enemigo deja de moverse y procede a realizar la animación de ataque básico. Una vez acabada la animación, si el jugador sigue en rango de ataque el enemigo volverá a atacar seguidamente y si el jugador ya no está a rango, el enemigo volverá a perseguirlo.

Algunos cambios respecto a la estructura del jugador son los siguientes. No contiene un nodo Camera2D ya que solo es necesario seguir al jugador con la cámara, no contiene un nodo CanvasLayer ya que no es deseable que la barra de vida esté en el HUD del jugador, el objetivo es que la vida de cada enemigo aparezca encima de su cabeza por separado y por último no contiene el nodo Area2D para detectar las puertas de las salas debido a que no es deseable que los enemigos puedan cambiar de sala, la idea es que se quedan en su habitación correspondiente.

## 6.3 Escena de las salas de la mazmorra

Todas las salas comparten la misma estructura así que se escogerá únicamente una (la sala main_room) y se explicará su estructura y qué más tipos de salas hay.

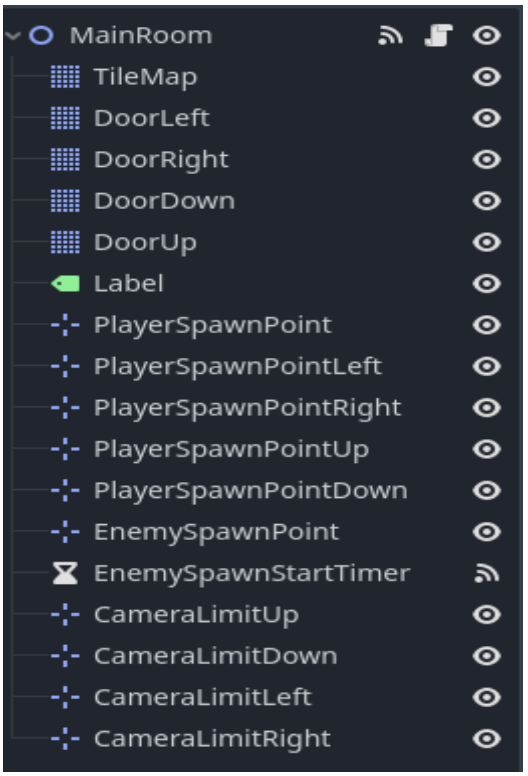


Figura 6.3.1: Captura de la estructura de los nodos de la sala main_room (Fuente: Elaboración propia)

El nodo raíz de las salas es tipo Node2D que básicamente representa un objeto 2D dentro del juego con su posición, orientación y escala, todos los nodos relacionados con el 2D heredan de este nodo. Las salas contienen un total de 5 nodos de tipo TileMap, un TileMap es una red de *tiles* (losas) que sirven para crear la disposición del mapa del juego, a estos *tiles* se les puede añadir colisiones que es el caso del proyecto. Se utilizan 5 de estos TileMaps de la siguiente manera. El primero con nombre TileMap es el que contiene la disposición de toda la sala y se encarga también de añadir la colisión a las paredes de la sala para que el jugador no pueda salir de ella. Los otros 4 restantes con los nombres DoorLeft, DoorRight, DoorDown y DoorUp se usan para la disposición de cada una de las puertas por las cuales el jugador podrá salir para cambiar de sala en la mazmorra. Se utiliza uno distinto para cada puerta y así poder distinguir de manera correcta la dirección por la cual sale el jugador y cambiar la sala acorde a esta puerta.

La escena también contiene 6 nodos de tipo Marker2D que son utilizados como puntos de aparición. Estos nodos Marker2D sirven simplemente como guía y se ven visualmente como una cruz en el editor de Godot, para el proyecto interesa principalmente su posición, ya que se utiliza para indicar el punto donde tienen que aparecer tanto los enemigos como el jugador. Para el jugador, la escena contiene un total de 5 puntos de aparición, el PlayerSpawnPoint que será donde aparece el jugador al empezar la partida en la sala main_room (solo la sala main_room contiene este punto de aparición). Los otros 4 puntos de aparición, PlayerSpawnPointLeft, PlayerSpawnPointRight, PlayerSpawnPointUp y PlayerSpawnPointDown, el jugador aparecerá en el punto de aparición correspondiente teniendo en cuenta la dirección de entrada a la sala.

Para los enemigos solo contiene un punto de aparición, el EnemySpawnPoint ya que el objetivo es que aparezcan los enemigos en el mismo sitio. También para los enemigos contiene un nodo de tipo

Timer, este nodo permite hacer una cuenta atrás, para el proyecto es utilizado para definir el tiempo que tardan los enemigos en aparecer cuando el jugador entra en la sala.

Para finalizar, la escena contiene otros 4 nodos de tipo Marker2D, CameraLimitUp, CameraLimitDown, CameraLimitLeft y CameraLimitRight, estos nodos sirven para que la cámara que sigue al jugador no sobrepase estos límites establecidos.

El resto de las salas comparten la misma estructura. Las salas existentes que hay en el juego son, main_room, normal_room, loot_room, key_room, dead_end y boss_room.

## 6.4 Algoritmo de generación procedimental de la mazmorra

En el proyecto, para guardar la información referente a la mazmorra se utilizará un *array*. En este array se guardarán instancias del *script* room_base.gd que es el que contiene la información base de las salas de la mazmorra. Cada una de las instancias de este *script* contendrá el tipo de sala(main_room, normal_room, loot_room, dead_end, key_room y boss_room) y el número de enemigos que tiene.

El algoritmo para desarrollar la mazmorra es el siguiente. Primero, se elige la semilla de generación (se elige de manera aleatoria) que es el número que inicializa la selección de números aleatorios, es decir, si se elige la misma semilla obtendremos la misma secuencia de números aleatorios. Después, se selecciona el tamaño (número de salas) que tendrá la mazmorra, este tamaño será un número aleatorio entre un valor mínimo y un valor máximo que asignará el desarrollador. Después, se agrega al *array* la primera sala que será la main_room, acto seguido se realizará un recorrido por todo el *array* de la mazmorra. En cada elemento (sala) se seleccionará 3 de las 4 posibles conexiones que puede tener (arriba, abajo, derecha e izquierda) y se conectarán salas a cada dirección seleccionada. Este proceso continuará hasta alcanzar el tamaño de la mazmorra que haya sido elegido.

Esta sería la base del algoritmo, pero hay varios puntos que se han tenido en cuenta. El objetivo de Divine Journey es acabar con el jefe final que se encuentra en la sala boss_room, para hacerlo primero el jugador debe encontrar la sala key_room para así hacer que cuando vaya a la boss_room aparezca el jefe. Por lo tanto, teniendo en cuenta esto, solo puede generarse una boss_room y una key_room, este concepto se aplica al algoritmo. Otro apartado a tener en cuenta es que el propósito de las salas dead_end como su nombre indica, es que sean un callejón sin salida. Por lo tanto, de una sala dead_end no pueden tener más de una conexión este punto se ha implementado en el algoritmo de generación . El último punto que se ha tenido en cuenta es que para hacer la sala boss_room más complicada de encontrar, esta sala tampoco genera nuevas conexiones como en el caso de la sala dead_end.

## 6.5. Gestión de animaciones

Las animaciones en Godot con los nodos AnimationPlayer y AnimationTree, aparte de permitir manejar las animaciones de manera más sencilla y proporcionar cambios fluidos entre animaciones, también permiten cambiar los valores de variables e incluso llamar funciones en *frames* de animación específicos. A continuación, se explicará qué son los nodos AnimationPlayer y AnimationTree, qué funcionalidades ofrecen y cómo se utilizan en el proyecto.

El nodo AnimationPlayer permite tener una librería de animaciones la cual puede ser utilizada para reproducir las animaciones. También permite definir el tiempo y el modo en el que se realiza una transición entre las animaciones.

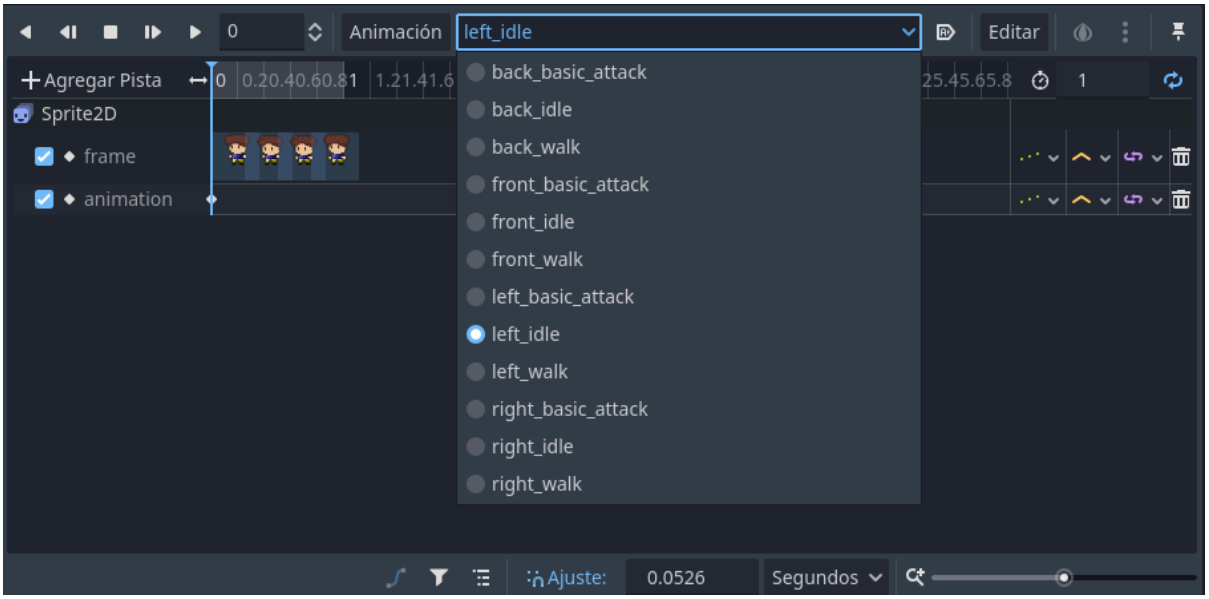


Figura 6.5.1: Captura de la pestaña Animación del editor de Godot (Fuente: Elaboración propia)

En la figura 6.5.1 se observan las pistas donde se coloca cada uno de los *frames* de la animación correspondiente y las distintas animaciones asociadas, en este caso, al jugador. La animación left_idle, es la animación que se reproduce cuando el jugador está inactivo y mirando en dirección a la izquierda. Para esta animación solo son necesarias las pistas de los *frames* y *animation*, ya que el objetivo es que no haga ninguna funcionalidad más aparte de reproducir la animación.

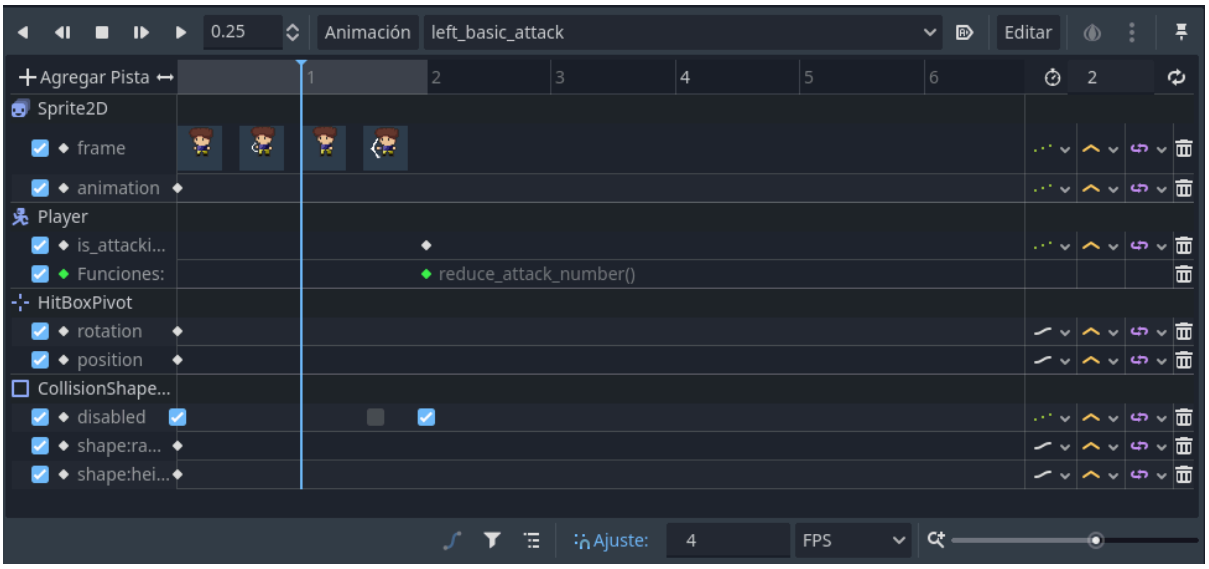


Figura 6.5.2: Captura Pestaña Animación del editor de Godot mostrando la animación left_basic_attack (Fuente: Elaboración propia)

En la figura 6.5.2 puede observarse que  la animación left_basic_attack contiene un número mayor de pistas que en la animación left_idle. Aparte de las pistas para los “frames” de la animación, también contiene varias pistas que controlan el comportamiento de diversas variables de la escena Player. A continuación, se explicarán algunas de estas pistas para entender su funcionamiento.

En el apartado Player de la figura 6.5.2 se observa una pista llamada is_attacking, está pista representa la variable del mismo nombre de la clase Player. Esta pista de is_attacking permite cambiar el valor de la variable en un momento de la animación determinado agregando una *key* (en la figura 6.5.2 esta *key* es el rombo blanco que está en la pista de is_attacking). En el contexto del proyecto, al final de la animación se cambia el valor de la variable a *false* ya que cuando acabe la animación el jugador ya no estaría atacando.

Otra variable que se controla con una pista es la variable disabled del CollisionShape2D del nodo Hitbox del jugador que controla si detecta o no colisiones. Primero, se asegura que esté desactivado al principio de la animación, ya que no es deseable que detecte colisiones en ese momento de la animación, entonces en el *frame* del impacto del golpe (el tercer *frame*) se activa la colisión. Al finalizar la animación se vuelve a desactivar esta colisión. Todo esto se consigue marcando cada momento de activación y desactivación cómo *keys* dentro de la pista.

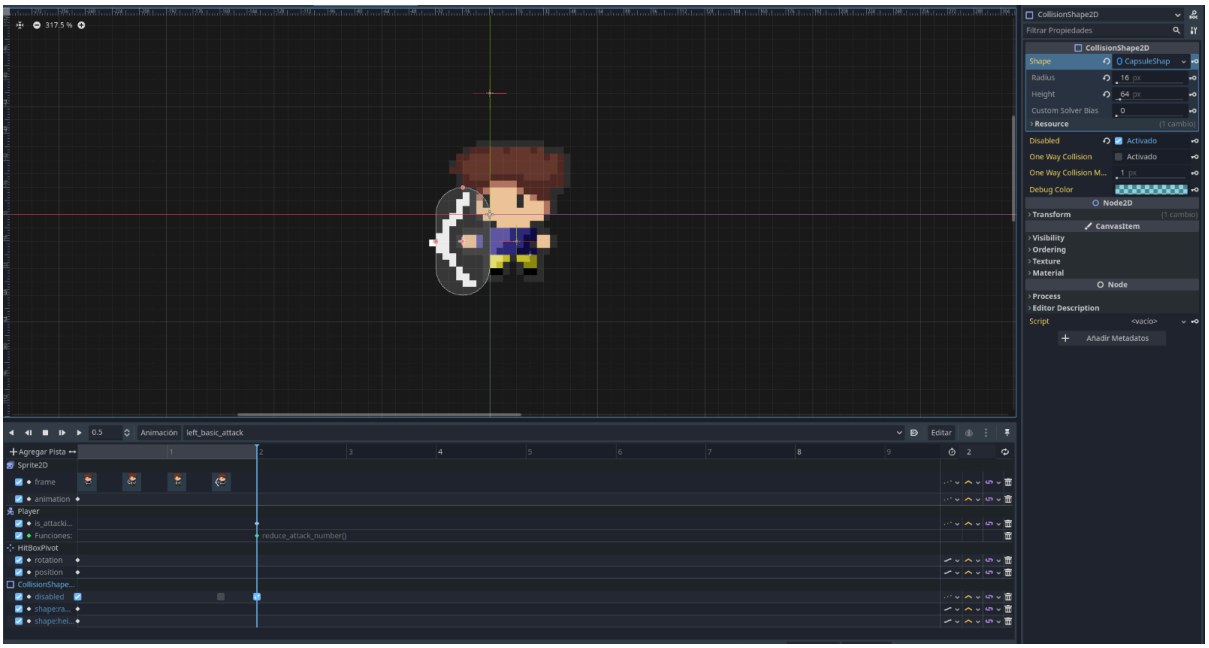

Figura 6.5.3: Captura del CollisionShape2D de la Hitbox del jugador (Fuente: Elaboración propia)

Por último, se realiza el mismo procedimiento para otras variables y parámetros que interese cambiar, como el tamaño y la forma del CollisionShape2D del Hitbox para que se adapte al “frame” del golpe. En la figura 6.5.3 se puede observar la colisión representada como una esfera estirada transparente con tonos grises.


Figura 6.5.4: Captura del Inspector de Godot del CollisionShape2D del jugador (Fuente: Elaboración propia)

La figura 6.5.4 muestra en la parte derecha el icono de una llave de color blanco, este icono es el que permite poner una *key* en una pista de animación para el AnimationPlayer y asignarle el valor correspondiente al que muestre el inspector cuando se reproduzca la animación.

Por otro lado, el nodo AnimationTree permite controlar cuándo y de qué manera reproducir las animaciones de un AnimationPlayer. El AnimationTree no tiene animaciones de por sí, si no que usa

las de un AnimationPlayer. Para controlar las transiciones entre animaciones se utiliza una *State Machine* (máquina de estados) donde cada estado será un tipo de animación (Idle, Walk y Attack para el jugador) y la transición entre estados dependerá de variables. A continuación, se observará con más detalle en el caso de la State Machine del jugador.

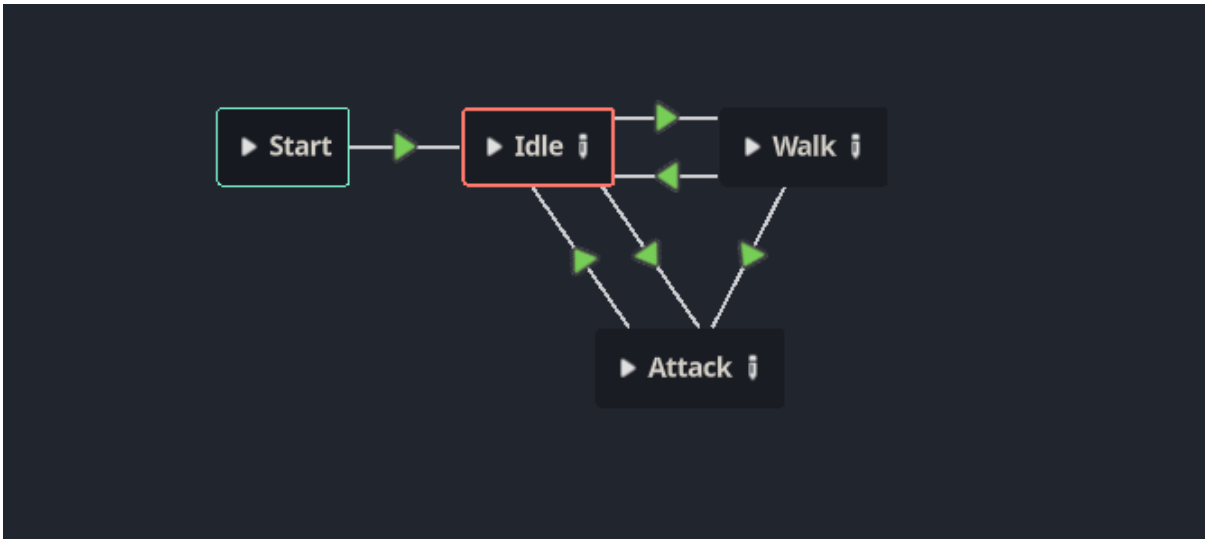


Figura 6.5.5: Captura del State Machine de la escena del jugador (Fuente: Elaboración propia)

Como puede observarse en la figura 6.5.5, la máquina de estados del jugador para las animaciones consta de un total de 3 estados de animación, Idle (estado inactivo, cuando el jugador no hace nada pasa a este estado), Walk (estado caminando, cuando el jugador está moviéndose pasa a este estado) y Attack (estado atacando, cuando el jugador ataca pasa a este estado). Se ha diseñado la máquina de estados de manera que se pueda realizar una transición del estado Idle a cualquiera de los otros dos estados, ya que el jugador puede pasar de estar quieto a caminar o de estar quieto a atacar directamente. Del estado Walk se puede realizar una transición también a cualquiera de los dos estados siguiendo la misma lógica que con el estado Idle. Cuando el jugador se está moviendo, puede pararse o puede atacar. Por último, del estado Attack solo se puede realizar una transición al estado Idle y de ahí pasar al estado Walk si el jugador está en movimiento.

Para determinar cuándo hacer la transición a cada estado se utilizan dos variables, la variable de tipo booleano is_walking, que representa si el jugador está caminando, cuando sea falsa se asumirá el estado Idle y cuando sea verdadera se asumirá el estado Walk. La otra es la variable también de tipo booleano is_attacking que representa si el jugador está atacando o no, cuando sea verdadera se asumirá directamente el estado Attack independientemente del valor de is_walking, cuando sea falsa entonces se podrá asumir los otros dos estados Idle o Walk dependiendo del movimiento del jugador.

Cada uno de los estados de la máquina de estados es un BlendSpace2D. Un Blendspace2D es un espacio virtual en dos dimensiones donde las animaciones son representadas como puntos y se realizan las transiciones entre cada una teniendo en cuenta el valor de un vector2D.

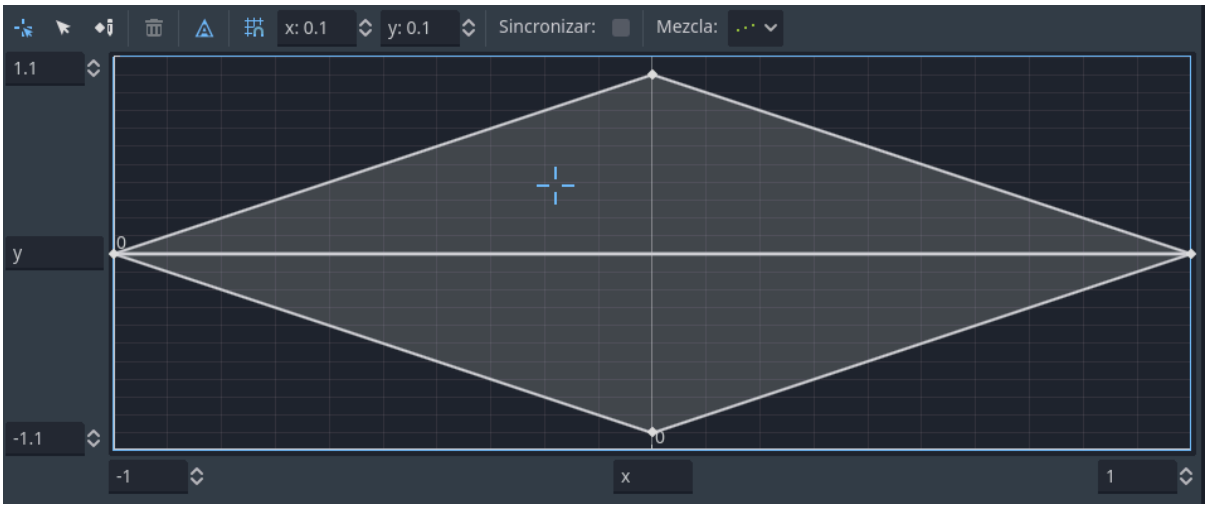


Figura 6.5.6: Captura del BlendSpace2D del estado Idle del jugador (Fuente: Elaboración propia)

Las animaciones son los puntos blancos que pueden observarse en la figura 6.5.6. Dependiendo del valor de un Vector2D que reciba, reproduce una animación o la otra. En el caso del proyecto el Vector2D que recibe el BlendSpace2D es el *input* que el jugador introduce en el juego. Este *input* representa la dirección del jugador y dependiendo de esta reproduciremos la animación acorde a la dirección actual.

Por último, la máquina de estados del enemigo es similar a la del jugador, pero no contiene el estado Idle ya que la idea es que los enemigos no estén nunca quietos y persigan constantemente al jugador.


Figura 6.5.7: Captura del State Machine de la escena del enemigo (Fuente Elaboración propia)

## 6.6 Gestión de colisiones

Como ha sido explicado en el apartado *3.1 Requisitos del videojuego*, para el sistema de combate del juego se utilizan principalmente dos tipos de escenas. Estas escenas representan la Hitbox (usada para infligir daño) y Hurtbox (usada para recibir daño). El sistema Hitbox y Hurtbox es la base para cualquier juego con un sistema de combate. A continuación, se explicará cómo se ha implementado en el videojuego.

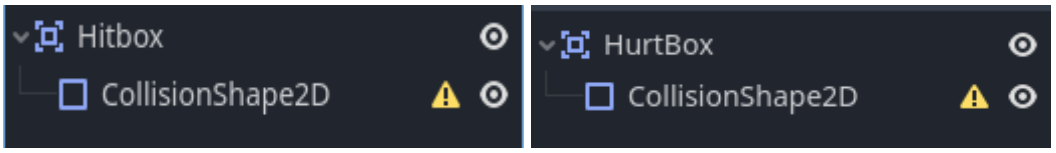


Figura 6.6.1: Captura de las estructuras de las escenas Hitbox y Hurtbox (Fuente: Elaboración propia)

Como puede observarse en la figura 6.6.1, las dos escenas son idénticas en estructura, las dos tienen como nodo raíz un nodo de tipo Area2D y un nodo hijo de tipo Collision Shape 2D que se encarga de dar una forma para detectar colisiones (de igual manera que funciona el nodo DetectDoor de la escena del jugador). Donde difieren es en las *Collision Layer* y *Mask Layer* que tienen asignados estos nodos Area2D, más adelante en este apartado se explicará cómo funcionan y de qué manera se han organizado en el proyecto. A continuación, se explicará cómo funciona el sistema Hurtbox y Hitbox sin explicar las *Collision Layer* y *Mask Layer*.

Tanto el jugador como los enemigos tienen una Hurtbox y una Hitbox (dependiendo del sistema de combate que se quiera implementar se pueden tener más de una de cada tipo), la Hurtbox se encarga de detectar la colisión con una Hitbox con la cual puede dañar al objetivo, ya sea el jugador o un enemigo. Cuando detecta esta colisión, manda una señal y ejecuta la función o acción que se le haya asignado, en el contexto del proyecto la acción que se ejecuta es la de reducir los puntos de vida del jugador o enemigo.

A continuación se explicarán las *Collision Layer* y *Mask Layer*. Las colisiones en Godot se gestionan mediante el sistema de *Collision Layer*. Cada objeto de colisión (Un CharacterBody2D, un CollisionShape2D o un TileMap) tiene dos propiedades, la *Collision Layer* y la *Collision Mask*. Hay un total de 32 *layers* (capas) con las que un objeto puede interactuar. La *Collision Layer* indica la capa o capas físicas en la que está el objeto con colisión y la *Collision Mask* indica las capas físicas donde el objeto escanea para encontrar colisiones.

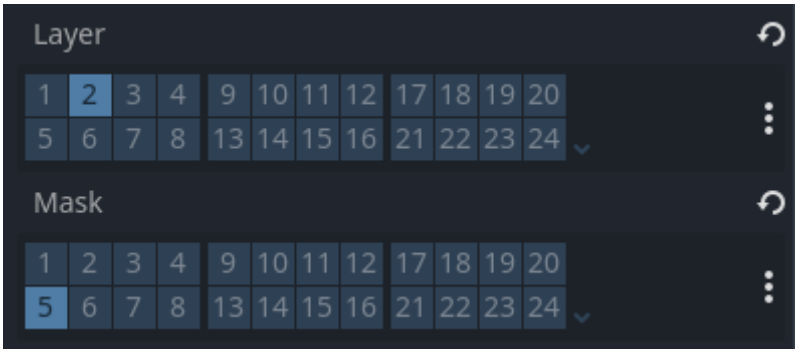


Figura 6.6.2: Captura de Collision Layer y Collision Mask del CharacterBody2D del jugador (Fuente: Elaboración propia)

Como puede observarse en la figura 6.6.2, para asignar las capas de colisión se seleccionan las capas deseadas para la asignación. Hay que tener en cuenta que cada capa representa un *bit* en el sistema binario, es decir, la capa 1 representa el primer bit, la capa 2 representa el segundo bit (con valor 2), la capa 3 representa el tercer bit (con valor 4) y así con las demás capas. Por lo tanto, para asignar las

capas mediante código la forma más fácil sería usando un número binario y poniendo 1s donde queramos que esté activa la capa. Por ejemplo, si se quiere asignar las capas 1, 3 y 5 el valor que se debe asignar es 0b10101 (valor en binario) que en decimal sería 21. Cada 1 del valor binario corresponde con una capa activada.

Estas capas pueden ser nombradas de manera diferente para poder distinguirlas y organizarlas mejor. A continuación, se explicará cómo se estructuran en el proyecto para cada uno de los objetos de colisión.

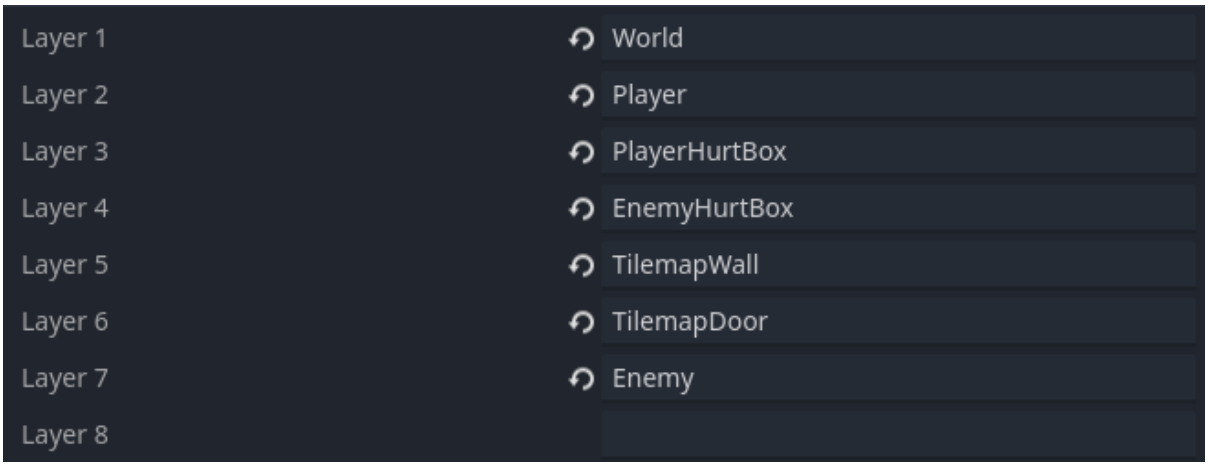


Figura 6.6.3: Nombre de las capas de colisión 2D del proyecto de Godot (Elaboración propia)

En la figura 6.6.3 se muestran los nombres de las 7 capas utilizadas para los objetos con colisión del proyecto. La primera capa World es para el mundo en general, se usa para que el jugador no caiga hacia abajo de la pantalla en caso de tener la gravedad activada, pero en el caso del proyecto no se utiliza esta capa. La capa Player es el *Collision Layer* que tiene el CharacterBody2D del jugador, ya que la capa física en la que existe, la capa PlayerHurtBox es la Collison Mask que tiene la HurtBox del jugador, debido a que escaneará en es capa física en busca de alguna colisión. Por otra parte, la capa EnemyHurtBox es la *Collison Mask* que tiene la Hurtbox del enemigo y escaneará colisiones en esa capa física. La capa TilemapWall es la *Collision Layer* que tienen las *tiles* de la pared del TileMap de todas las salas, debido a que es la capa física en la que existirán para que el jugador no las pueda atravesar. Por otra parte, la capa TilemapDoor es la *Collision Layer* que tiene los Tilemap de las puertas de las salas y esta será la capa que el jugador escaneará para poder detectar las colisiones con una puerta. Finalmente, la capa Enemy sigue la misma lógica que la capa Player pero para los enemigos. Esta capa es la *Collision Layer* que tiene el characterBody2D de los enemigos.

## 6.7 Optimización para cargar las escenas

Como se ha explicado en el apartado *3.1 Requisitos del videojuego*, uno de los requisitos del videojuego es el cambio entre salas para poder explorar la mazmorra. Esto se realiza cargando una escena nueva al cambiar de sala. Está manera de exploración es eficiente de cara a poder organizar la estructura de la mazmorra ya que puede tenerse en un *array* que contenga cada una de las salas de la mazmorra como se ha explicado en el apartado *6.4 Algoritmo de generación procedimental de la mazmorra*. El problema encontrado para esta implementación es que no se puede realmente cargar las escenas de las salas cada vez que se realiza un cambio, ya que esto provocaría que el jugador tenga que esperar un tiempo de carga elevado cada vez que quiera ir a una nueva sala. Este enfoque sería muy poco eficiente y provocaría una disminución de la jugabilidad. Para solucionar este problema se ha creado una nueva escena que se carga nada más abrir el juego y se encarga de cargar las escenas en segundo plano mientras le muestra al jugador el porcentaje de carga que lleva. Una vez cargadas las escenas se guardan en las variables de escenas del Singleton RoomInformationGlobal.

Para cargar las escenas en segundo plano se utilizará los hilos de ejecución o *threading*. El *threading* es una técnica que permite ejecutar simultáneamente varias operaciones en el mismo espacio de proceso, por lo tanto, se utilizarán para poder ejecutar simultáneamente la pantalla de carga utilizando el hilo principal de ejecución y los hilos secundarios de ejecución para cargar las escenas.

En Godot para cargar las escenas utilizando *threading* se utiliza el método load_threaded_request() de la clase ResourceLoader. Este método permite cargar recursos utilizando diferentes hilos de ejecución, en el caso del proyecto se utiliza para cargar las escenas. Para saber el estado de carga de las escenas, el método “load_threaded_get_status()” también de la clase ResourceLoader, proporciona esta información. Con estos métodos se puede cargar las escenas en segundo plano y a la vez mostrar al jugador el progreso de carga.

Una vez cargadas las escenas y guardadas en las variables del Singleton RoomInformationGlobal, se utilizan estas variables para cargar las escenas de forma casi instantánea cuando el jugador realiza el cambio entre las salas de la mazmorra.

# 7. Resultados

En este capítulo se observarán los resultados finales obtenidos después del proceso de desarrollo y las pruebas realizadas a usuarios para comprobar la correcta obtención de los requisitos del videojuego.

## 7.1 Testing

Una vez preparada la primera versión de Divine Journey, se preparó una sesión de *testing* con usuarios para comprobar las funcionalidades del videojuego.

Los perfiles de usuarios que se han buscado para el *testing* son: Jugadores experimentados, usuarios con experiencia previa en juegos del género *roguelike*. Jugadores novatos, usuarios sin experiencia significativa en juegos *roguelike*. El último perfil es, desarrolladores y diseñadores, usuarios con conocimientos técnicos sobre el desarrollo de un videojuego de este género, proporcionando así *feedback* sobre el diseño del videojuego.

Durante la sesión de *testing* se encontraron diversos *bugs* (fallos) pero en general la implementación de las funcionalidades había salido bien. A continuación se explicarán algunos de los *bugs* encontrados y las ideas para corregirlos para el desarrollo futuro. También se explicarán algunas recomendaciones de funcionalidades que los usuarios comentaron que estarían interesados en verlas implementadas en un futuro.

Los *bugs* más significativos detectados son:

- Al entrar a una sala las puertas están bloqueadas (tienen colisión y el jugador no puede atravesarlas), al derrotar todos los enemigos de la sala las puertas se deberían desbloquear (quitar la colisión) menos las que no tengan ninguna conexión. Resulta que al derrotar a todos los enemigos, en todas las salas menos la main_room se desbloquean todas las puertas independientemente de si tienen conexión o no. La idea para corregir este *bug* es revisar el cambio de colisión de los TileMap que componen las puertas.

- Al hacer clic varias veces seguidas en un corto periodo de tiempo en el botón izquierdo del ratón, la animación de atacar del jugador se puede bloquear en el último *frame*. Cuando ocurre esto se bloquea todo el sistema de animaciones y se muestra constantemente el último *frame* de la animación de atacar en vez de la animación correspondiente. La idea para solucionar esto es poner un tiempo de espera para poder realizar un ataque consecutivo, es decir, que para poder volver a esperar haya que esperar un determinado tiempo.

Las funcionalidades más significativas que han recomendado los usuarios son:

- Los jugadores se han desorientado bastante por la mazmorra, piden alguna manera de orientarse mejor. Han recomendado un mapa para orientarse por la mazmorra. Esta funcionalidad ya estaba pensada implementarla en la versión final del juego, así que pasará a ser prioritaria para el desarrollo futuro.

- Los jugadores han pedido más formas de atacar a los enemigos, recomiendan que haya más tipos de ataques. Se pensaron anteriormente dos *user stories* para implementar más tipos de ataques, una *user story* para realizar ataques básicos con distintas armas y una *user story* para implementar habilidades para atacar a los enemigos.

Estos *bugs* y recomendaciones se tendrán en cuenta para un futuro desarrollo del videojuego.

## 7.2 Resultados finales

En este apartado se mostrarán los resultados finales para la primera versión del videojuego Divine Journey.

Al abrir el juego y después de cargar las salas, el jugador podrá ver el menú principal con el que podrá interactuar para iniciar una partida o salir del juego si así lo prefiere.


Figura 7.2.1: Menú principal del juego Divine Journey (Fuente: Elaboración propia)

Personaje principal y enemigo

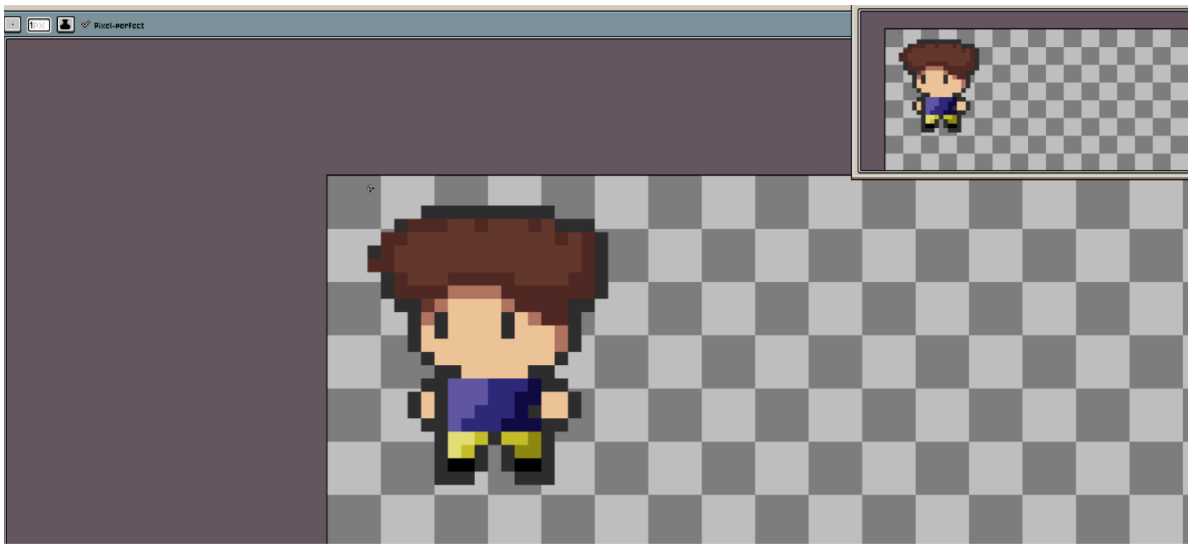

Figura 7.2.2: Diseño del personaje realizado en Aseprite (Fuente: Elaboración propia)

Figura 7.2.2: Diseño del enemigo “Goblin” hecho en Aseprite (Elaboración propia)

Salas de la mazmorra

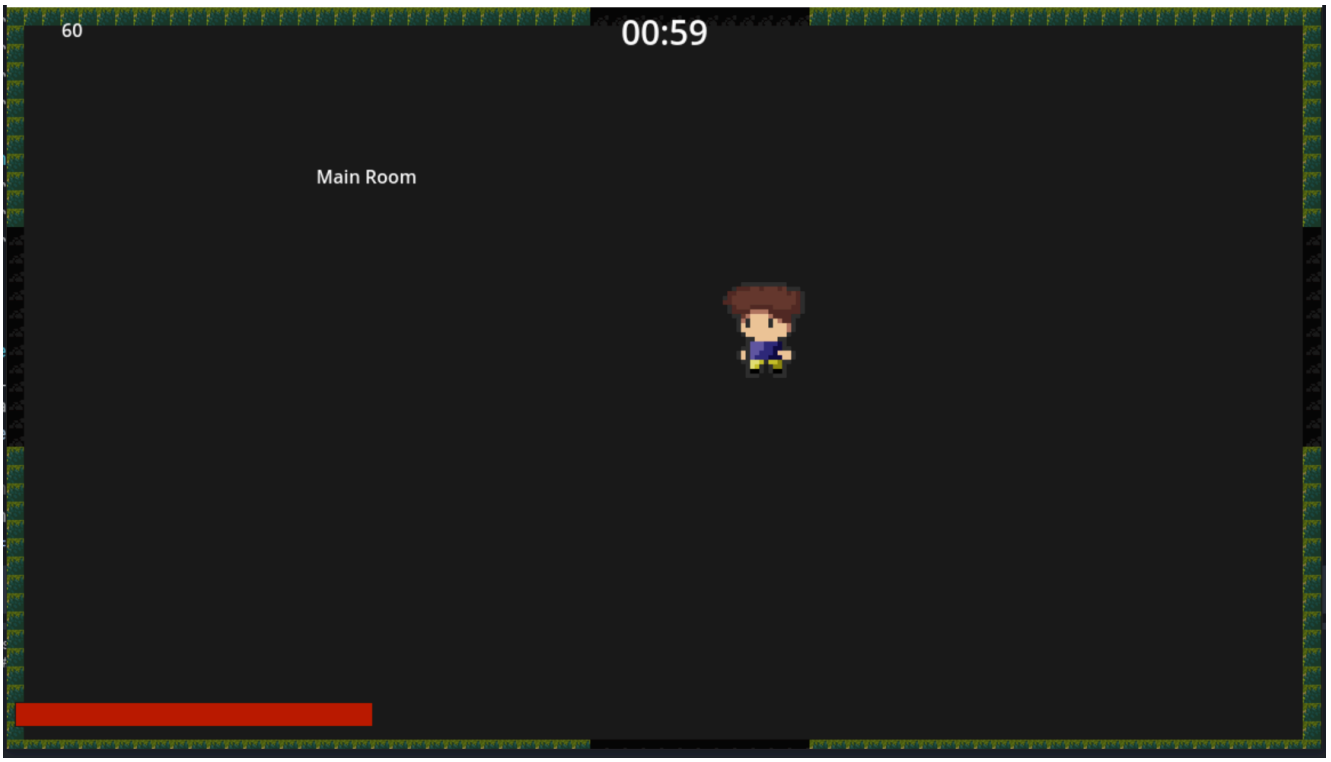


Figura 7.2.3: Diseño de la sala main_room (Fuente: Elaboración propia)

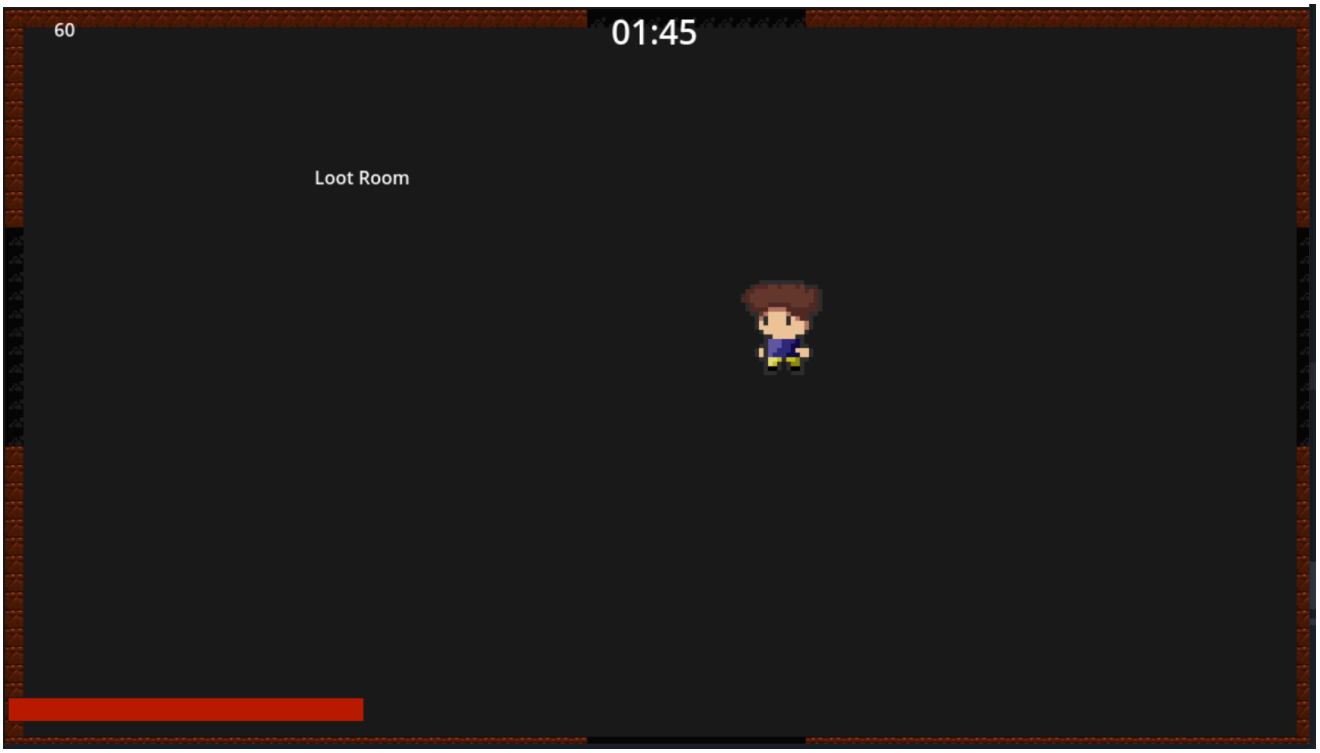


Figura 7.2.4: Diseño de la sala loot_room (Fuente: Elaboración propia)

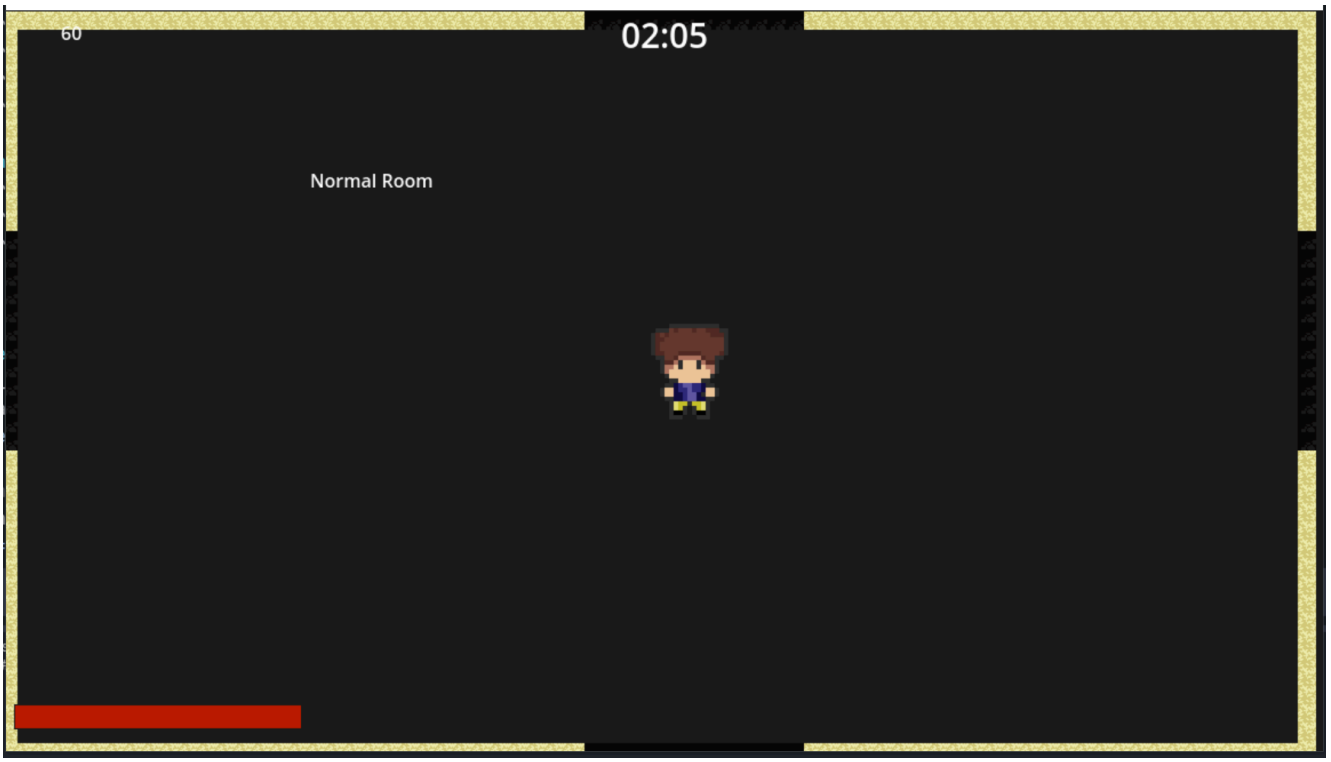


Figura 7.2.5: Diseño de la sala normal_room (Fuente: Elaboración propia)

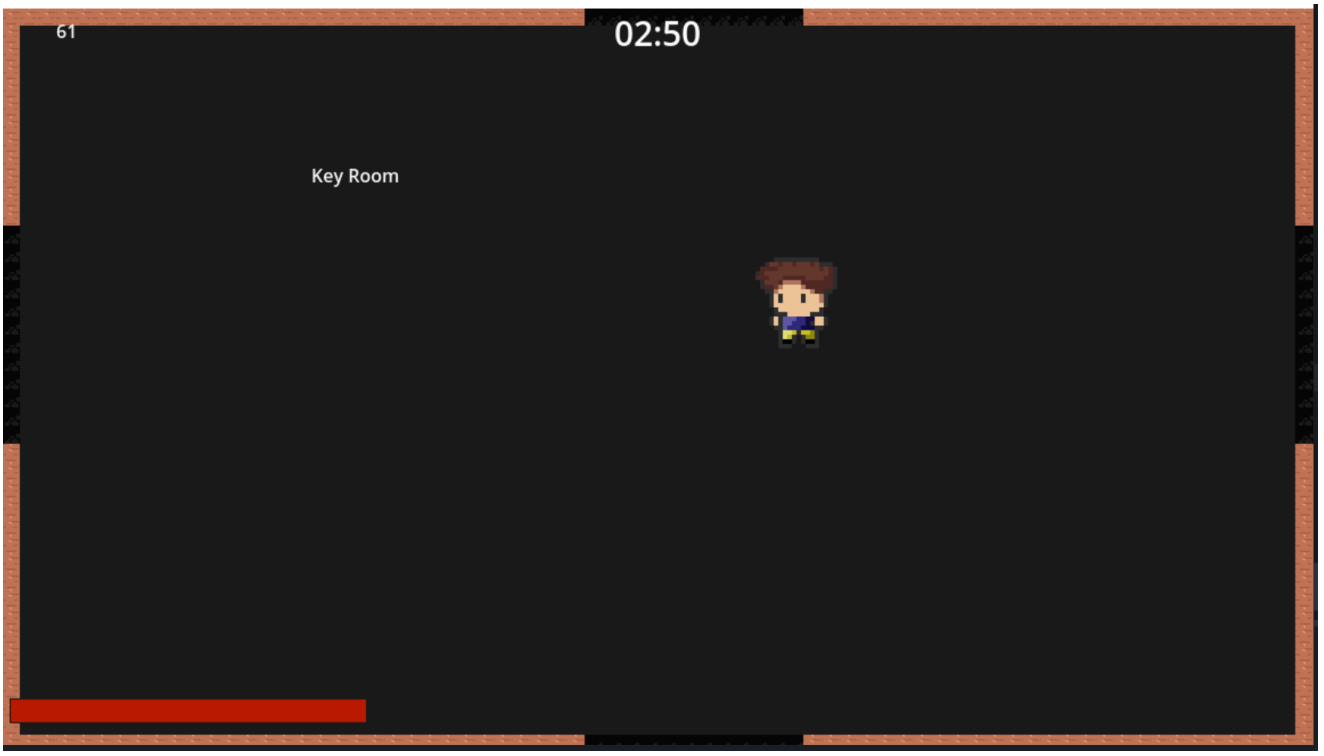


Figura 7.2.6: Diseño de la sala key_room (Fuente: Elaboración propia)

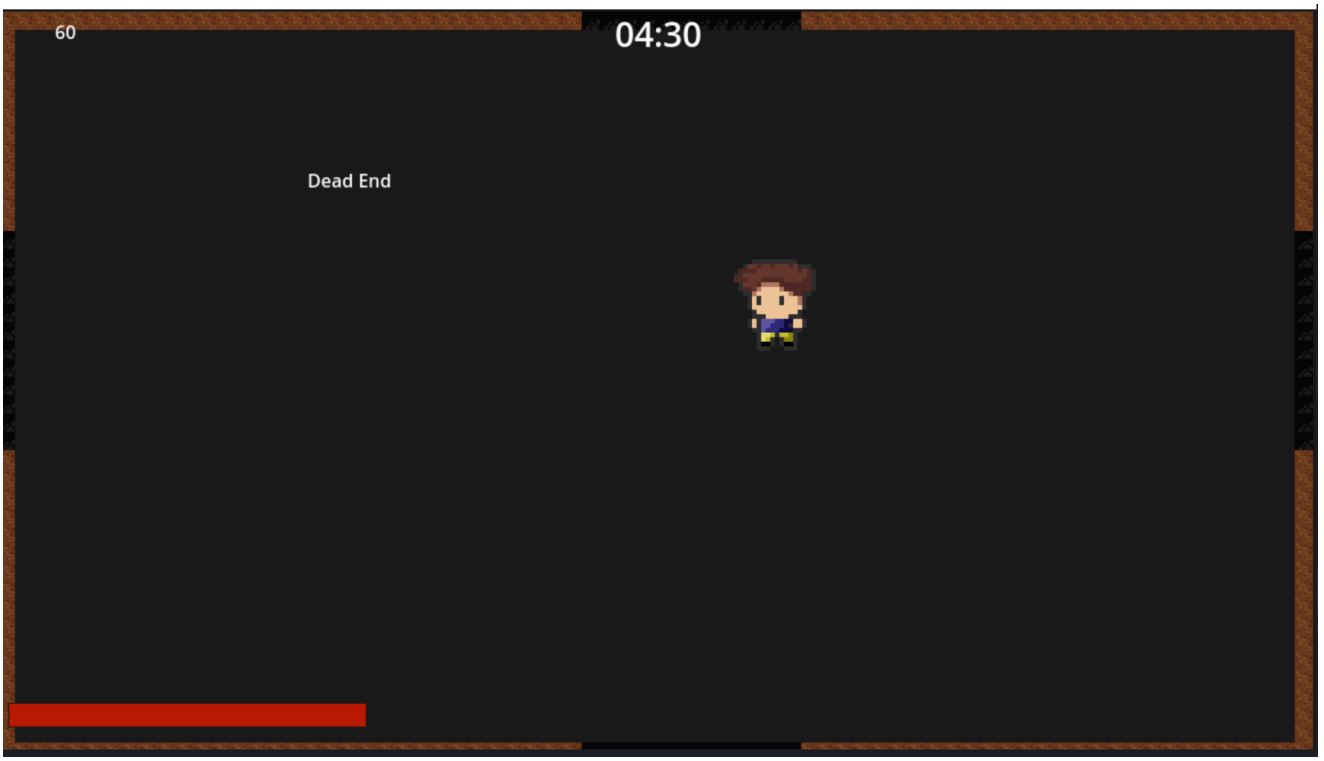


Figura 7.2.7: Diseño de la sala dead_end (Fuente: Elaboración propia)

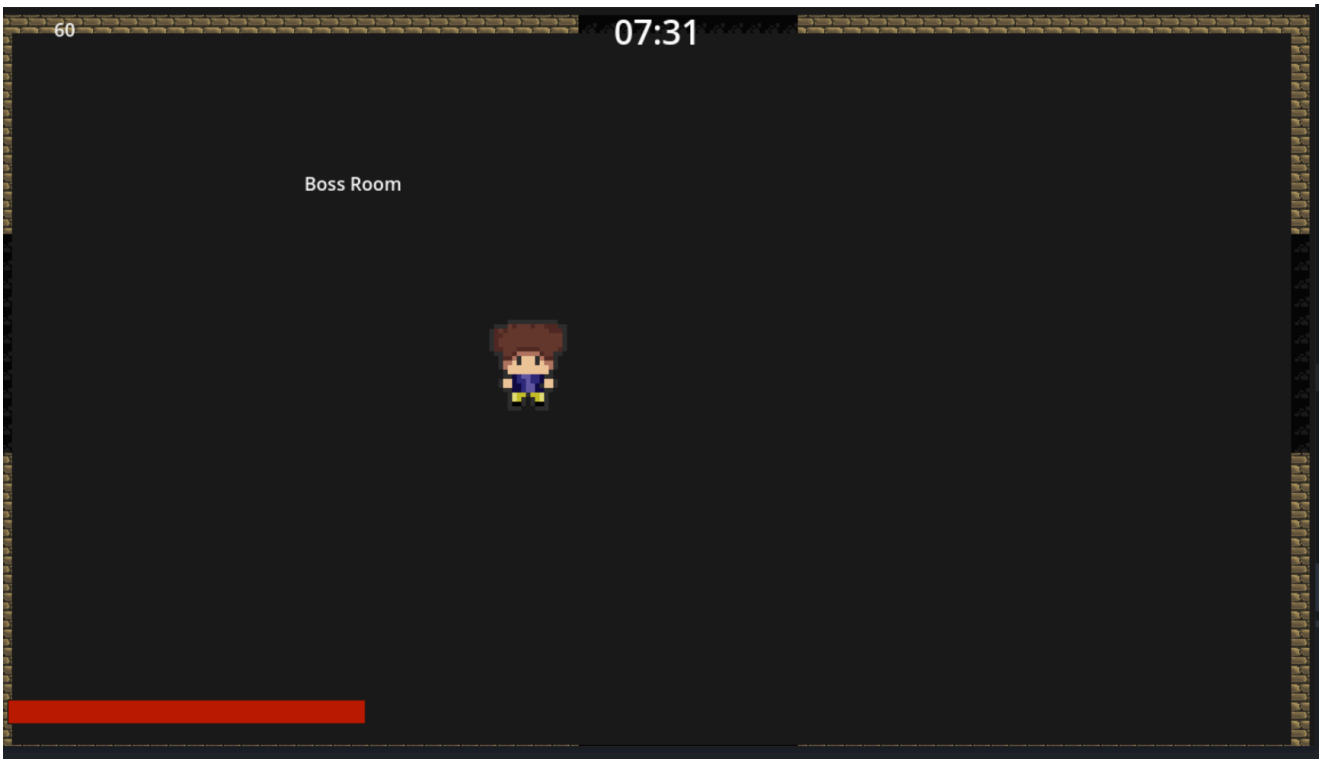


Figura 7.2.3: Diseño de la sala boss_room (Fuente: Elaboración propia)

Desarrollo de una partida

Al hacer clic en el botón *play* del menú principal se carga la escena principal y el jugador comienza la partida. En cada sala aparece un enemigo que ataca al jugador, no se desbloquearán las puertas de la sala hasta vencer al enemigo. El jugador debe encontrar la sala del jefe final (boss_room) y vencerlo para ganar, pero para que aparezca el enemigo final en la sala boss_room primero se debe encontrar la sala key_room. Una vez encontrada la sala key_room, al entrar en la sala boss_room, aparecerá el enemigo final y el jugador podrá vencerlo. Después de la derrota del jefe final, el jugador volverá al menú principal.

# 8. Conclusiones

En este proyecto se ha conseguido implementar una primera versión del videojuego Divine Journey. Esta primera versión cumple con todos los objetivos mencionados en el apartado *1.2 Objetivos* menos uno. El sistema de sonido no se ha llegado a implementar debido a los cambios durante el desarrollo y la falta de tiempo de implementación.

La realización de este proyecto ha consolidado los conocimientos adquiridos durante el grado de Ingeniería Informática junto con el aprendizaje de nuevas tecnologías. Ha sido necesario aprender desde 0 el uso de Godot y el proceso para desarrollar un videojuego, pero gracias a las habilidades en diseño de software y conocimientos en Python ha resultado más ágil el aprendizaje del motor gráfico Godot y su lenguaje de programación GDScript, el cual comparte similitudes con Python.

Por lo tanto, se considera que la primera versión del videojuego Divine Journey cumple con gran parte de los requisitos iniciales y está preparada para ser enseñada a un inversor para su desarrollo futuro.

## 8.1 Trabajo futuro

Como se ha explicado en el capítulo *8. Conclusiones*, el sistema de sonido no se ha podido implementar debido a los cambios y problemas surgidos durante el desarrollo, por lo tanto, el desarrollo futuro se centrará en implementar las bases de este sistema.

En el apartado *3.3 User stories* se mostró diversas *user stories* que no se considerarón prioritarias para la primera versión de Divine Journey pero fueron pensadas para agregar nuevas funcionalidades al videojuego para alcanzar la versión final. Por lo tanto, si el inversor diera su visto bueno al proyecto, el desarrollo futuro se centrará también en implementar todas las funcionalidades explicadas en las *user stories* restantes. Estas funcionalidades serían:

- Implementar la interacción con el botón *Settings* del menú principal para que el jugador pueda cambiar la configuración del juego. Algunos aspectos que podría cambiar el jugador son, el volumen de la música y los efectos de sonido, la resolución de la pantalla, entre otros.

- Implementar la funcionalidad de desplazamiento rápido para que el jugador pueda esquivar a los enemigos, ofreciendo así una mayor profundidad al combate.

- Agregar diferentes tipos de armas y ataques básicos correspondientes a cada arma.

- Agregar más estadísticas (propiedades) al personaje principal y enemigos y la posibilidad de visualizar estas estadísticas por parte del jugador.

- Agregar potenciadores para que el jugador pueda mejorar sus estadísticas.

- Implementar un sistema de habilidades para dar más variedad de ataques al jugador para combatir a los enemigos.

- Agregar un mapa de la mazmorra para ayudar al jugador a orientarse a la hora de explorarla.

La versión final del juego estaría pensada para lanzarse en la tienda de venta *online* de juegos en PC para que esté disponible para todos los usuarios interesados.

# 9. Bibliografía


[1] Kanban Tool. (s.f.). *Metodología Kanban*. [Consultado el 17 de febrero de 2024]. Recuperado de https://kanbantool.com/es/metodologia-kanban

[2] Proyectos Ágiles. (s.f.). *¿Qué es Scrum?.* [Consultado el 17 de febrero de 2024]. Recuperado de https://proyectosagiles.org/que-es-scrum/

[3] Fernández, Rosa. (2024). *Cuota de mercado mundial de los sistemas operativos*. Statista. [Consultado el 8 de junio de 2024]. Recuperado de https://es.statista.com/estadisticas/576870/cuota-de-mercado-mundial-de-los-sistemas-operativos/

[4] The Good Gamer. (s.f.). *Desarrollo de un videojuego: ¿En qué etapas se divide?*. [Consultado el 22 de Febrero]. Recuperado de https://thegoodgamer.es/desarrollo-de-un-videojuego-en-que-etapas-se-divide/

[5] Keith, Clinton. (2010). *Agile game development with Scrum*. Addison-Wesley Professional. ISBN 9780321670311

[6] GDQuest. (n.d.). *Entity-Component pattern*. [Consultado el 2 de marzo de 2024]. Recuperado de https://www.gdquest.com/tutorial/godot/design-patterns/entity-component-pattern/

[7] Godot Engine. (n.d.). *Documentation*. [Consultado a lo largo del proyecto]. Recuperado de https://docs.godotengine.org/en/stable/index.html

[8] Fandom. (n.d.). *Risk of Rain 2 Wiki*. [Consultado el 19 de febrero de 2024]. Recuperado de https://riskofrain2.fandom.com/wiki/Risk_of_Rain_2_Wiki

[9] Fandom. (n.d.). The Binding of Isaac Wiki. [Consultado el 19 de febrero de 2024]. Recuperado de https://bindingofisaac.fandom.com/es/wiki/The_Binding_of_Isaac

# 10. Anexos

## 10.1 Glosario

·**API:** Siglas para Application Programming Interface, conjunto de funciones y procedimientos que permite que sus funcionalidades se puedan reutilizar en otras aplicaciones.

·**Bug:** Error o fallo en el código que causa un comportamiento no deseado o inesperado en el juego.

·**Dash:** En videojuegos, movimiento rápido del personaje, la funcionalidad de este depende del videojuego. La más común es acercarse rápidamente a un enemigo.

·**FPS:** Siglas para Fotogramas por Segundo, la cantidad de fotogramas que el juego procesa en cada segundo.

·**HUD:** Siglas para Head-Up Display, es la interfaz para comunicar al jugador información a través de la pantalla, como la vida del jugador, el tiempo transcurrido, la vida de los enemigos, etc.

·**Indie:** Juego independiente desarrollado por un estudio relativamente pequeño y no por una gran empresa.

·**Sprint:** Intervalo de tiempo el cual se utiliza para desarrollar unas determinadas funcionalidades de un producto o software hasta que sean utilizables.

·**Sprite:** En programación de videojuegos, conjunto de imágenes que representan un personaje o un objeto del juego.

·**US:** User stories.

·**User stories:** Historias de los usuarios que ayudan a explicar de manera más concreta los requisitos.

## 10.2 Manual para generar ejecutable del código

En este apartado se explicará el proceso para crear un ejecutable del videojuego Divine Journey en caso de que haya algún problema con el ejecutable proporcionado.

Primero se debe descargar Godot Engine del siguiente enlace: https://godotengine.org/download/archive/. Se recomienda elegir la versión 4.1.1-stable debido a que Divine Journey se desarrolló utilizando esa versión.

Una vez descargado Godot, se debe descargar y descomprimir el .zip del código fuente. Acto seguido se debe abrir Godot Engine y abrir el proyecto descargado.

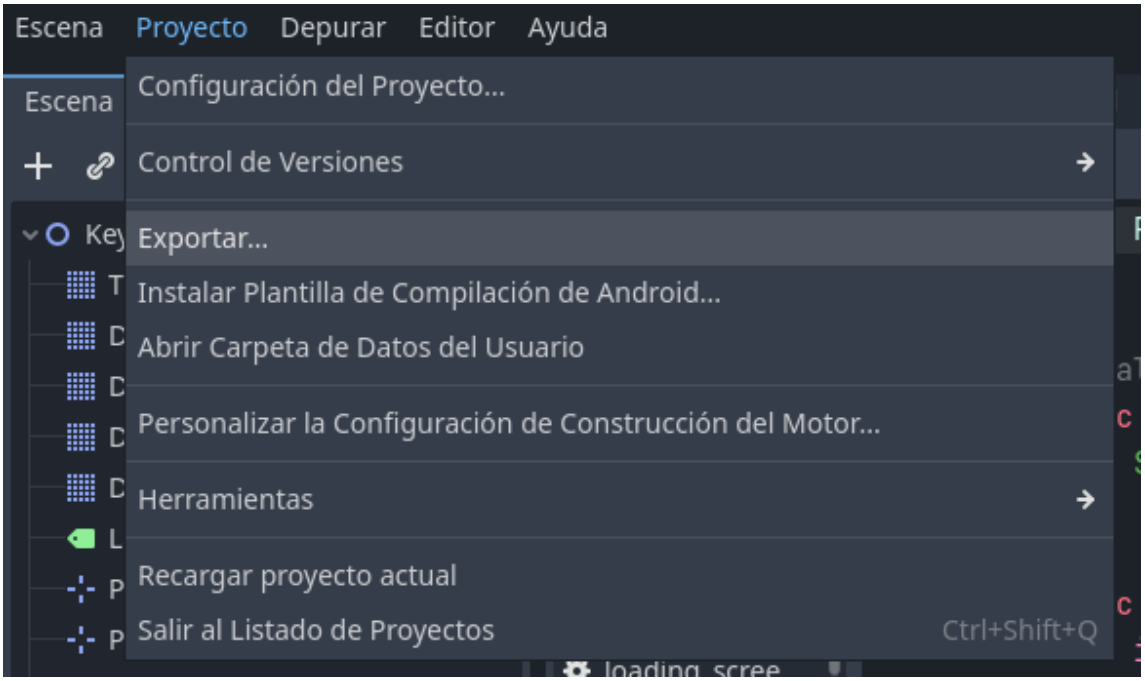


Figura 10.2.1: Captura de las opciones de proyecto de Godot (Fuente: Elaboración propia)

Como se muestra en la figura 10.2.1, se debe hacer clic en el botón de exportar.

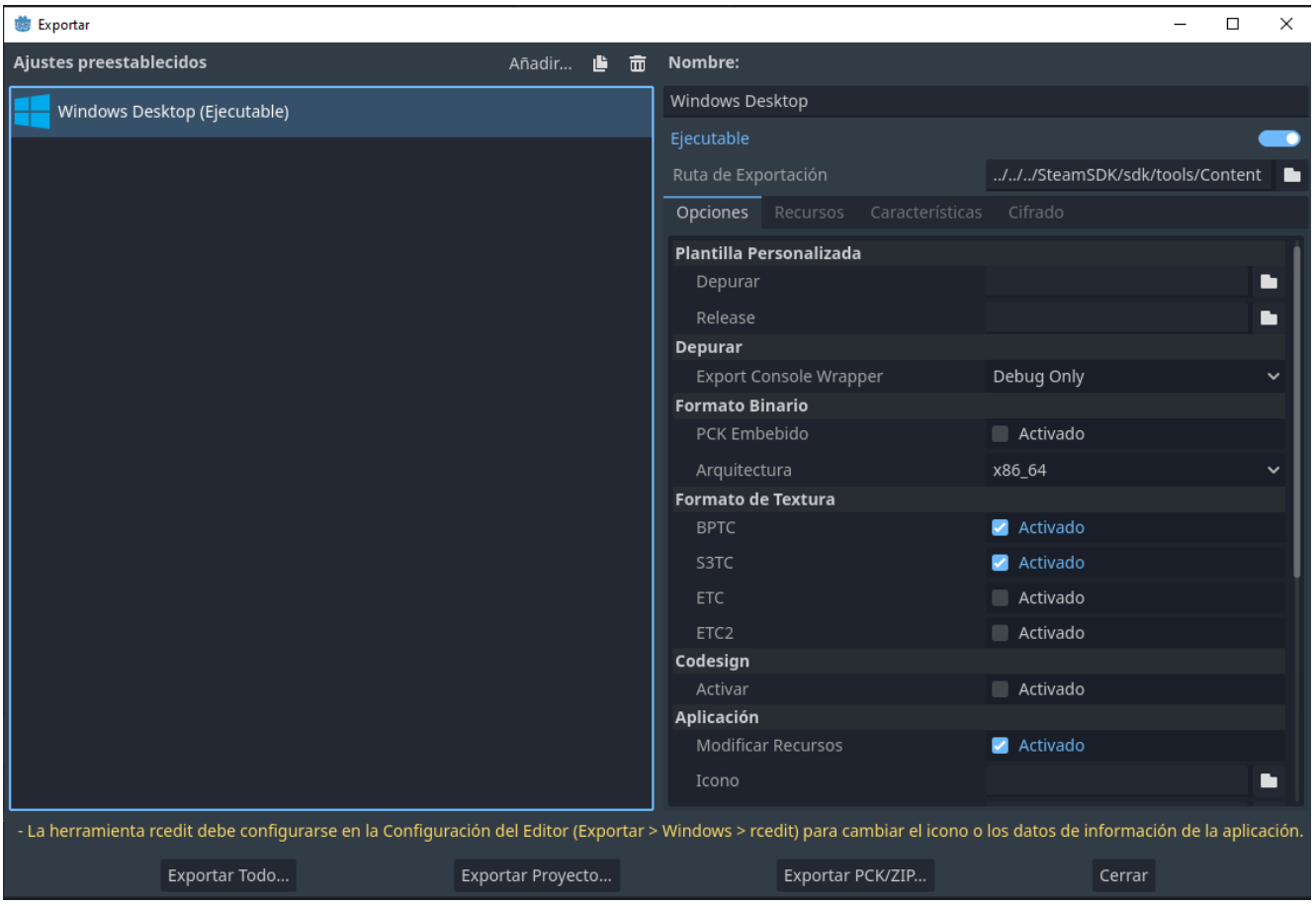


Figura 10.2.2: Captura de opciones de la pestaña exportar del proyecto (Fuente: Elaboración propia)

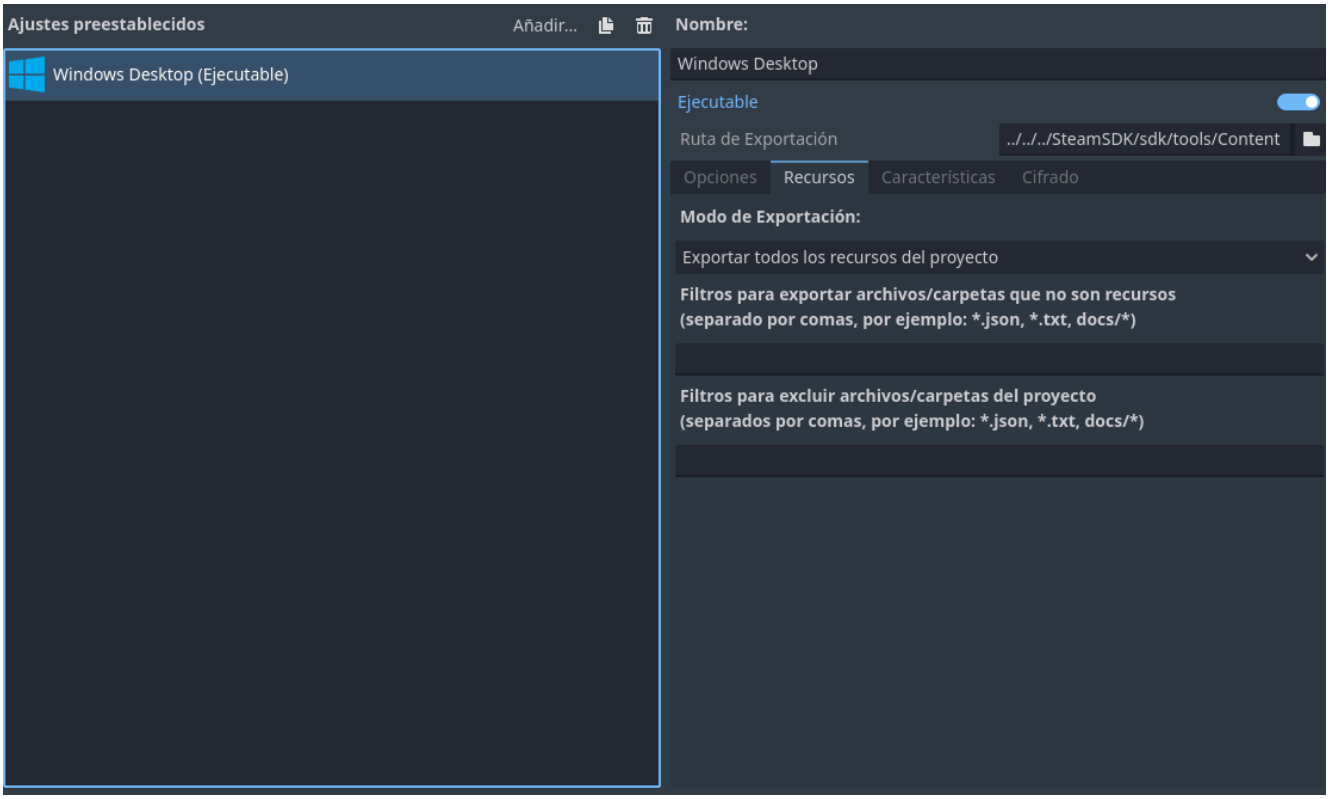


Figura 10.2.3: Captura de recursos de la pestaña exportar del proyecto (Fuente: Elaboración propia)

Se deben dejar las opciones mostradas en las figuras 10.2.2 y 10.2.3. Una vez terminado hay que hacer clic en exportar todo. Esto nos creará el ejecutable en la ruta que se haya especificado en la parte de *Ruta de Exportación*.