UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

ONIROS, un videojuego metroidvania 2D desarrollado en Godot Engine

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: García Pérez, Mariano

Tutor/a: Molina Marco, Antonio

CURSO ACADÉMICO: 2023/2024

AGRADECIMIENTOS

A mi tutor Antonio, por descubrirme la rama de Ingeniería del Software y por su guía para pulir este trabajo lo máximo posible.

A mis padres, por su apoyo incondicional y la confianza que han depositado en mí.

A Carlos, Pablo y Alonso, por aguantar todo mi estrés este último año.

A Javier y Jary, por su dedicación, recomendaciones y por ser mi *testers* de confianza.

A María, por su cariño y paciencia durante el tiempo de realización de este trabajo.

Y a Rocky, la mayor fuente de inspiración de mi vida. Este trabajo no existiría sin tí.

Muchas gracias.

RESUMEN

Este trabajo presenta el desarrollo de un videojuego 2D tipo metroidvania, titulado *ONIROS*, utilizando el motor *Godot Engine* y un estilo visual pixel-art. El proyecto se centra en la creación de una primera iteración del juego, que incluye las mecánicas básicas de jugabilidad como exploración, combate y progresión del personaje. El objetivo principal es consolidar conocimientos en programación orientada a objetos, aplicando un enfoque modular y reutilizable que permitirá futuras expansiones y mejoras del juego. Además, se busca optimizar el rendimiento y asegurar la compatibilidad con diferentes dispositivos de entrada. Este trabajo no solo demuestra habilidades técnicas en ingeniería de software y diseño de videojuegos, sino que también contribuye al portafolio profesional del autor en un mercado competitivo.

Palabras clave: ingeniería del software; videojuego; metroidvania; pixel-art; Godot Engine

ABSTRACT

This work presents the development of a 2D metroidvania-type video game titled *ONIROS*, utilizing the *Godot Engine* and a pixel-art visual style. The project focuses on creating the first iteration of the game, including basic gameplay mechanics such as exploration, combat, and character progression. The primary objective is to consolidate knowledge in object-oriented programming, applying a modular and reusable approach that will allow for future expansions and improvements to the game. Additionally, the aim is to optimize performance and ensure compatibility with various input devices. This work not only demonstrates technical skills in software engineering and video game design but also contributes to the author's professional portfolio in a competitive market.

Keywords: software engineering; video game; metroidvania; pixel-art; Godot Engine

ÍNDICE

ÍNDICE DE ILUSTRACIONES

CAPÍTULO 1:
INTRODUCCIÓN

En este primer apartado, se va a describir brevemente los fundamentos de este trabajo de fin de grado, haciendo hincapié en cuáles son las motivaciones para elaborar este proyecto, cuáles son los objetivos por alcanzar con el desarrollo de este proyecto y qué relación tiene con lo que se ha aprendido a lo largo de la titulación. Por último, se proporcionará un breve esquema que permita al lector de este documento conocer en que fases se divide y qué puntos se van a tratar, de manera que pueda facilitar la lectura y comprensión.

1.1. Motivación del TFG

La idea principal de este TFG surge a raíz de la necesidad de documentar de algún modo la primera iteración de este proyecto software, como más adelante se describirá en el apartado 1.2 sobre los objetivos.

La motivación principal para desarrollar este videojuego nace del deseo del autor de poner a prueba y consolidar sus conocimientos en programación orientada a objetos, explorando un proyecto que se aleje de las experiencias previas realizadas durante la titulación. Además, este proyecto representa una oportunidad clave para construir un porfolio en el sector del videojuego, permitiendo al autor demostrar la experiencia adquirida en el desarrollo de videojuegos e ingeniería del software. Este porfolio no solo servirá como una vitrina de las habilidades técnicas desarrolladas, sino también como una carta de presentación efectiva en el competitivo mercado laboral.

El impulso definitivo para llevar a cabo este proyecto se dio tras el anuncio a finales de 2023 de un videojuego del mismo género y temática que la idea concebida por el autor: un juego de plataformas del subgénero *metroidvania* (cuyo significado se explicará en capítulos posteriores), centrado en la temática de la pérdida. El videojuego en cuestión, *Tales of Kenzera: ZAU* (Vandal, 2023), capturó la atención del autor y reforzó su determinación de desarrollar una propuesta similar, aplicando su visión única y aprovechando la oportunidad de crear un proyecto que no solo demuestre sus habilidades técnicas, sino que también resuene emocionalmente con los jugadores.

1.2. Objetivos del TFG

El objetivo principal de este trabajo sería crear una primera versión del videojuego con las características básicas de jugabilidad habilitadas, con la exploración, el combate contra enemigos o la obtención de puntos de experiencia. Como resultado, además de la propia versión de demostración se dispondrá de documentación general de la primera iteración de este proyecto. Esto servirá para plantear un primer esquema general de los requisitos necesarios, las metodologías seguidas en el desarrollo y los hitos alcanzados hasta la fecha de cierre del documento, pudiendo hacer una previsión de las tareas a realizar para futuras iteraciones del desarrollo.

En lo que respecta a objetivos específicos destacaría cuatro:

- Desarrollar una primera versión del videojuego con la funcionalidad básica y libre de fallos, en la medida de lo posible.

- Implementar todas las mecánicas de los enemigos diseñados facilitando su reutilización en iteraciones posteriores.
- Minimizar el tiempo de carga e inicio del juego (no mayor a 30 segundos) y el tiempo de transición entre escenas (no mayor a 10 segundos).
- Implementar controles cómodos e intuitivos que favorezcan la usabilidad y la experiencia de usuario.

Para poder cumplir estos objetivos, se utilizará como modelo de proceso de software el ciclo de vida clásico o en cascada: análisis, diseño, codificación y pruebas. Dado que en este caso solo es desarrollará una primera iteración, no se realizará la fase de despliegue.

1.3. Relación con lo aprendido en la doble titulación

Todos los contenidos presentados en este documento, aunque están adaptados a la temática de un videojuego, siguen los estándares presentados en la asignatura de *Ingeniería del Software*.

Por otro lado, este proyecto se ha elaborado con un lenguaje de programación integrado en el propio motor gráfico que se emplea para el diseño del juego (como se explicará con mayor detalle en el capítulo 3, sobre el estado del arte), que comparte semejanzas sintácticas con el lenguaje de programación Python. Para desarrollarlo, he tenido que hacer un proceso de formación previa en este lenguaje (Torres, 2024), el cual ha sido más sencillo de cursar gracias a los conocimientos adquiridos sobre sintaxis de lenguajes interpretados y orientados a objetos que se estudia en la asignatura *Lenguajes, tecnologías y paradigmas de la programación*.

Por último, pero no menos importante, para elaborar los menús de interacción, los manejadores de eventos y comprender el funcionamiento de un motor gráfico, se han aplicado algunos de los conocimientos obtenidos en la asignatura *Interfaces Persona Computador*, como por ejemplo los principios de diseño de interfaces y la estructuración escena-nodo, o el modelo vista controlador.

1.4. Estructura del TFG

Además de este capítulo, este TFG cuenta con ocho capítulos más:

- En el capítulo 2 se examinan los **productos similares** existentes en el mercado, destacando tres videojuegos clave que son referencia para el proyecto. Además, se analizan las **tecnologías disponibles** para el desarrollo del juego, con un enfoque en los **motores gráficos** y sus características, y se justifica la **elección de motor** para el desarrollo del videojuego.
- En el capítulo 3 se analizan el concepto del videojuego, su **sinopsis**, las **características principales** del juego, su **género**, el **propósito** y **público objetivo**, y el **estilo visual** que se planea utilizar. También se discute el **alcance** del proyecto y se describe cómo se implementará una primera iteración técnica para probar las mecánicas del juego.
- En el capítulo 4 describe el **marco de desarrollo** utilizado en el proyecto, que incluye la **metodología basada en el modelo incremental** debido a las limitaciones de equipo. Además, se detallan las **herramientas empleadas**,

- En el capítulo 5 se presentan los requisitos funcionales del videojuego, incluyendo el flujo de juego, los casos de uso, y los controles que el jugador utilizará. También se detallan los requisitos no funcionales, que son esenciales para asegurar la calidad, rendimiento, y usabilidad del juego.
- El capítulo 6 aborda la arquitectura del videojuego, los patrones de diseño utilizados, como el **modelo MVC**, y los tipos de programación empleados, como la **programación orientada a objetos y a eventos**. Se describen además los principales **bucles de juego** y la integración de estos elementos en la estructura del proyecto.
- El capítulo 7 aborda la implementación del código del proyecto, comenzando con una visión general de los nodos más utilizados y su configuración en Godot. Posteriormente, se realiza un análisis detallado de las escenas principales, explicando la función de cada nodo y cómo contribuyen al juego. Además, se describen los métodos de los scripts asociados a cada escena, detallando su funcionamiento y su papel en el flujo del juego.
- En el capítulo 8 se detallan las pruebas realizadas para garantizar el funcionamiento del proyecto. Se abordan pruebas de escenas instanciables y no instanciables, verificando su correcta manipulación y estabilidad, así como pruebas de compatibilidad para asegurar que el juego funcione bien en diferentes plataformas. Estas pruebas son fundamentales para validar la robustez y versatilidad del juego antes de su lanzamiento.
- Y, por último, en el capítulo 9, se reflexiona sobre el desarrollo del proyecto, resaltando los hitos y logros principales. También identifica áreas de mejora y posibles soluciones, además de explorar características para futuras iteraciones. Se concluye con un balance de lo aprendido y el cierre de la versión de demostración, estableciendo las bases para el desarrollo futuro.

CAPÍTULO 2:
ESTADO DEL ARTE

En el presente capítulo se analizará el contexto actual del mercado de videojuegos, centrándose en dos aspectos fundamentales para el desarrollo del proyecto: los productos similares ya existentes y las tecnologías que se emplean para su creación. La comprensión del estado del arte es esencial para identificar las oportunidades y desafíos que presenta el desarrollo de un nuevo videojuego, así como para asegurar que el proyecto se alinee con las tendencias actuales y aproveche las mejores prácticas del sector.

2.1. Análisis de productos similares del mercado

Para poder desarrollar e introducir con éxito el juego en el mercado, uno de los pasos más importantes a realizar es el análisis de la competencia. Tras un análisis exhaustivo de juegos similares, se observa como destacan principalmente tres:

- *Hollow Knight*: es un videojuego *metroidvania* en 2D desarrollado y publicado por Team Cherry en 2017. Este videojuego (Carrión, 2022), aclamado por crítica y jugadores, es considerado uno de los mayores exponentes del género. Se caracteriza por:
    - Intricado Diseño de Niveles: Hollow Knight ofrece un vasto mundo interconectado, lleno de caminos y secretos que recompensan la exploración.
    - Desafío Equilibrado: El juego presenta una dificultad bien ajustada, ofreciendo un reto satisfactorio que recuerda a los clásicos de plataformas.
    - Estética Única: Su estilo artístico sombrío, acompañado de una música inmersiva, crea un ambiente melancólico y cautivador.
    - Narrativa Profunda: A través de la exploración y la interacción, los jugadores descubren una historia enigmática y rica en detalles.
    - Progresión Adictiva: La curva de aprendizaje permite mejorar habilidades y obtener nuevas capacidades, incentivando la exploración y la rejugabilidad

Ilustración 1. Imagen promocional de Hollow Knight. Fuente: Nintendo, 2024

- *Blasphemous*: es un videojuego *soulsvania* 2D desarrollado y publicado por el estudio español *The Game Kitchen* en 2019. Este videojuego (Cano, 2020), considerado uno de los mejores videojuegos españoles de la historia, se caracteriza por:
    - Diseño de Niveles: Blasphemous se destaca por un diseño de niveles meticulosamente elaborado, donde la exploración es clave. El juego presenta un mundo interconectado lleno de secretos, caminos ocultos y áreas que requieren habilidades específicas para ser accesibles.
    - Dificultad y Desafío: El juego ofrece un desafío considerable, combinando plataformas precisas y combates exigentes que requieren habilidad y paciencia. Es un juego que recompensa la perseverancia y la destreza del jugador.
    - Estética Visual y Atmosférica: Blasphemous impresiona con su estilo artístico inspirado en el arte religioso y la imaginería oscura. Los gráficos en 2D son detallados y evocan un ambiente lúgubre que contribuye a la inmersión en su mundo gótico.
    - Narrativa y Ambientación: La narrativa del juego es rica en simbolismo y está profundamente enraizada en la cultura y la religión españolas, creando una historia que es tanto intrigante como perturbadora. Los jugadores descubren la historia a través de fragmentos de diálogos y descripciones, lo que fomenta la exploración y la interpretación personal.
    - Sistema de Combate: Blasphemous cuenta con un sistema de combate preciso y satisfactorio, que se basa en ataques cuerpo a cuerpo y habilidades especiales. El combate es desafiante y requiere que los jugadores dominen los patrones de los enemigos y jefes para avanzar.
    - Progresión y Personalización: Los jugadores pueden mejorar sus habilidades y personalizar su estilo de juego mediante la adquisición de nuevas habilidades, objetos y mejoras. Este sistema de progresión incentiva la rejugabilidad y la exploración.

Ilustración 2. Imagen promocional de Blasphemous. Fuente: Valve Corporation, 2024a

- *Ori and the Blind Forest*: es un videojuego *metroidvania* 2D desarrollado y publicado por *Moon Studios* en 2015. Este videojuego (VidaExtra, 2015), destacado especialmente por su dirección artística, se caracteriza por lo siguiente:
    - Diseño Artístico: Ori and the Blind Forest destaca por su impresionante estilo visual, con escenarios dibujados a mano que crean una atmósfera mágica y envolvente.
    - Jugabilidad Fluida: El juego combina de manera magistral elementos de plataformas y metroidvania, con controles precisos que permiten una experiencia de juego dinámica y satisfactoria.
    - Narrativa Emocional: La historia del juego es conmovedora y está profundamente entrelazada con la experiencia de juego, transmitiendo un fuerte mensaje emocional desde el inicio.
    - Dificultad Desafiante: A pesar de su apariencia encantadora, Ori and the Blind Forest ofrece un desafío considerable, especialmente en sus secciones de plataformas que requieren precisión y habilidad.
    - Banda Sonora: La música del juego es otro de sus puntos fuertes, acompañando perfectamente a la narrativa y elevando la experiencia emocional y estética del juego.

Ilustración 3. Imagen promocional de Ori and the Blind Forest. Fuente: DLCompare, s.f.

2.2. Análisis de tecnologías del mercado

Para poder desarrollar un videojuego, generalmente es necesario emplear un motor gráfico. Un motor gráfico (Aller, 2024) es un software que se utiliza en el desarrollo de videojuegos y otras aplicaciones visuales, permitiendo la creación y gestión de gráficos en tiempo real. Este tipo de software facilita la integración de gráficos 2D y 3D, la física, la iluminación y otros elementos visuales, brindando a los desarrolladores las herramientas necesarias para diseñar mundos interactivos y dinámicos. Los motores gráficos son fundamentales para la creación de videojuegos modernos, ya que optimizan el proceso de desarrollo y garantizan una experiencia visual coherente y fluida.

A continuación, se presentan las características de los diferentes motores gráficos disponibles en el mercado que son más adecuados para el desarrollo de videojuegos de plataformas o videojuegos en 2D.

2.2.1. Unity

Unity (Canle, 2023) es un motor gráfico reconocido por su versatilidad y accesibilidad, lo que lo convierte en una opción popular tanto para desarrolladores principiantes como para profesionales. Algunas de sus características destacadas incluyen la capacidad de crear experiencias en 2D y 3D, un entorno de desarrollo intuitivo con soporte para múltiples plataformas, y una comunidad activa que facilita el aprendizaje y la resolución de problemas. Unity también ofrece herramientas avanzadas para gráficos, física y animación, y su compatibilidad con lenguajes de programación como C# lo hace una opción robusta para el desarrollo de videojuegos y aplicaciones interactivas.

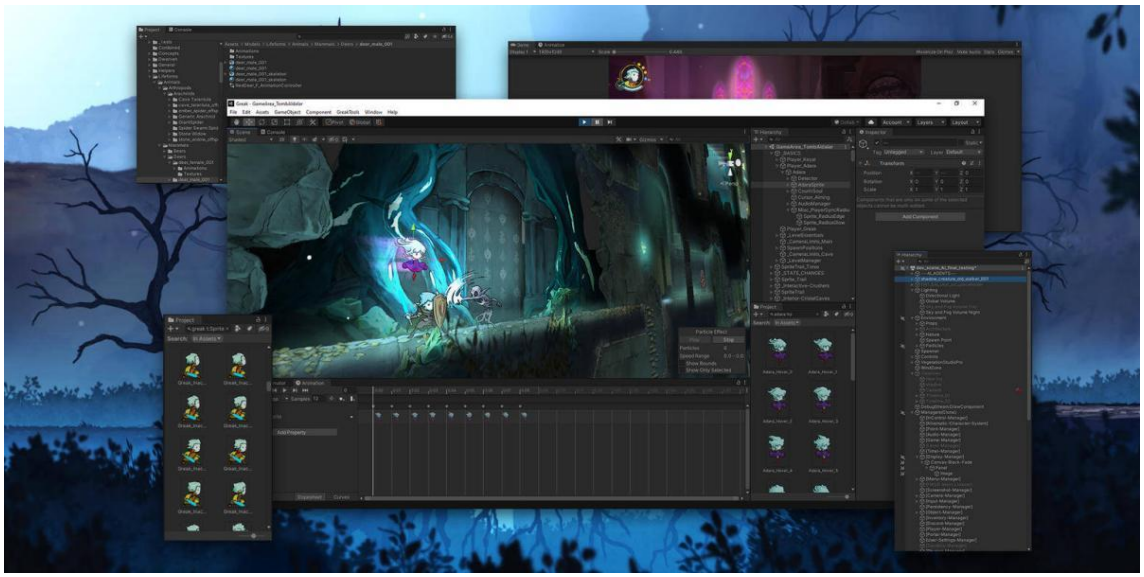


Ilustración 4. UI del motor Unity. Fuente: Gómez, 2023

2.2.2. Game Maker Studio

GameMaker (Código Vanguardia, s.f.-a) es una herramienta de desarrollo de videojuegos que permite a los usuarios crear juegos en 2D de manera sencilla y eficiente. Es conocido por su facilidad de uso, lo que lo hace accesible tanto para desarrolladores novatos como para experimentados. Entre sus principales características destacan:

- Interfaz Intuitiva: GameMaker cuenta con una interfaz de usuario amigable que facilita la creación y gestión de proyectos.

- Sistema de Arrastrar y Soltar: Permite a los desarrolladores crear juegos sin necesidad de conocimientos avanzados en programación, utilizando un sistema de arrastrar y soltar para definir la lógica del juego.
- Lenguaje de Programación Integrado (GML): Para los usuarios que desean mayor control, GameMaker incluye su propio lenguaje de programación, el GameMaker Language (GML), que permite crear scripts más complejos y personalizados.
- Compatibilidad Multiplataforma: GameMaker facilita la exportación de juegos a múltiples plataformas, incluyendo Windows, Mac, Android, iOS, y consolas, entre otras.
- Recursos Integrados: Ofrece una biblioteca de recursos gráficos y sonoros que los desarrolladores pueden utilizar para acelerar el proceso de creación.
- Extensibilidad: Los usuarios pueden expandir las capacidades de GameMaker mediante la integración de extensiones y plugins desarrollados por la comunidad o por ellos mismos.

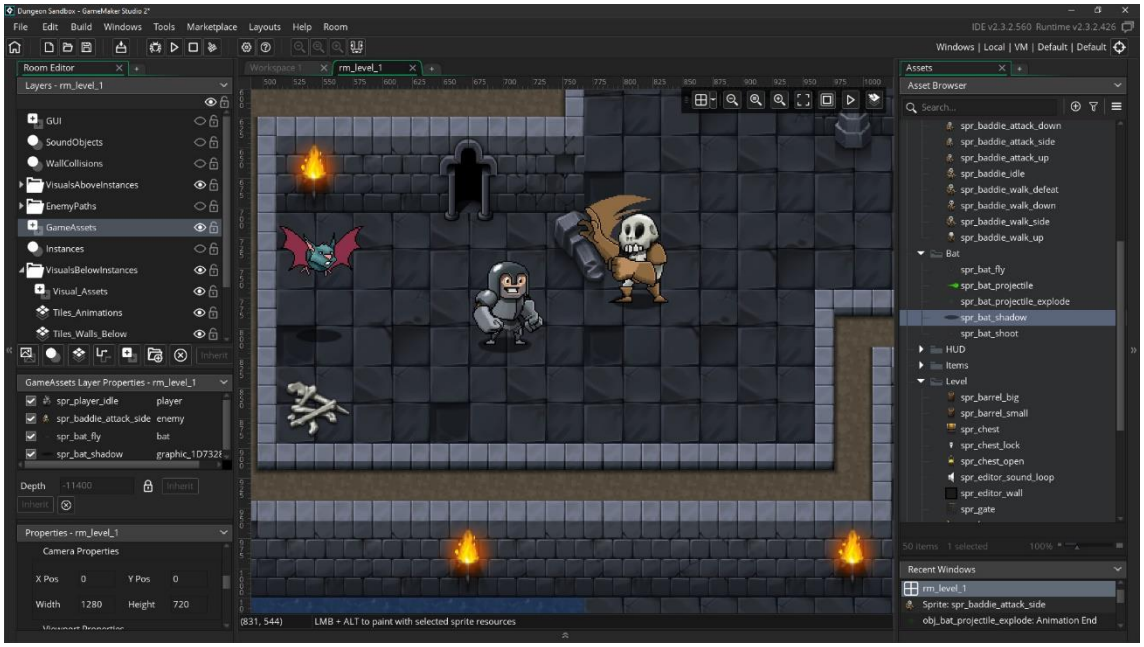


Ilustración 5. UI de Game Maker Studio. Fuente: Valve Corporation, 2024b

2.2.3. Godot Engine

Godot Engine (Código Vanguardia, s.f.-b) es un motor gráfico de código abierto y gratuito, ampliamente utilizado para el desarrollo de videojuegos en 2D y 3D. Algunas de sus características más destacadas incluyen:

- Códigos Abierto y Multiplataforma: Godot permite a los desarrolladores acceder y modificar su código fuente, además de ofrecer soporte para múltiples plataformas como Windows, macOS, Linux, Android, iOS, HTML5, y más.
- Escena y Sistema de Nodo: Godot organiza el contenido del juego en una estructura de escenas y nodos, lo que permite una gestión modular y flexible de los componentes del juego.

- Lenguaje de Programación GDScript: Incluye su propio lenguaje de scripting, GDScript, diseñado específicamente para facilitar el desarrollo rápido y eficiente de videojuegos. También soporta otros lenguajes como C#, C++, y VisualScript.
- Editor Visual: Cuenta con un editor visual intuitivo que facilita el diseño y la construcción de juegos sin necesidad de programación avanzada.
- Rendimiento Optimizado: Aunque es un motor ligero, Godot ofrece un rendimiento competitivo tanto en gráficos 2D como en 3D.
- Soporte Activo y Comunidad: Godot tiene una comunidad activa y un soporte continuo que proporciona actualizaciones y mejoras regulares.

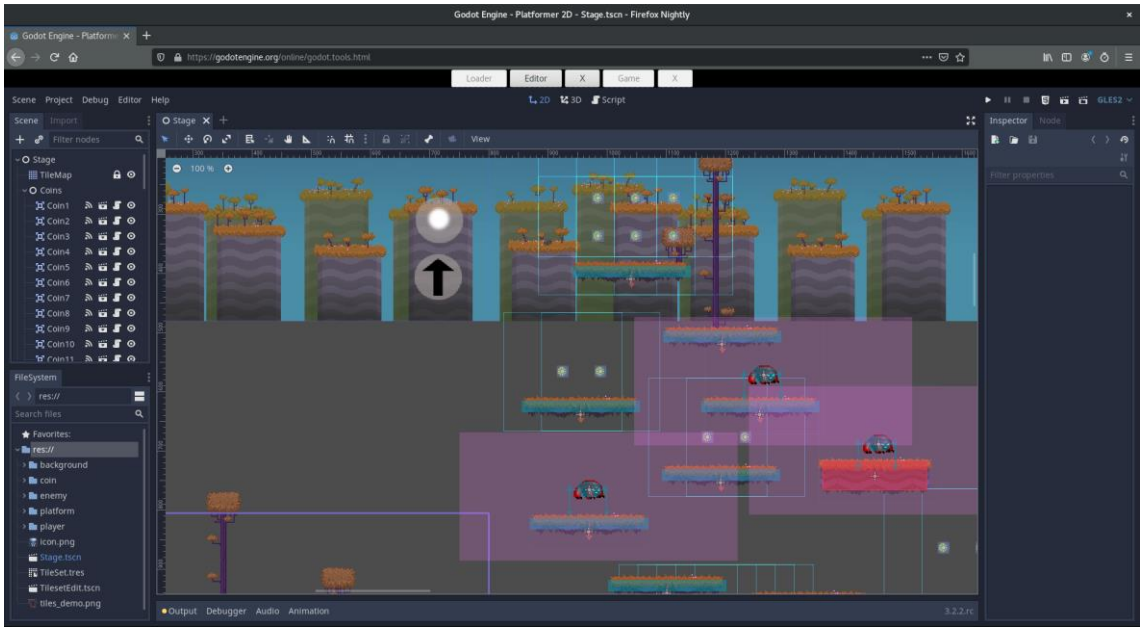


Ilustración 6. UI del motor Godot Engine. Fuente: Alessandrelli, 2022

2.3. Propuesta y motor gráfico seleccionado

A diferencia de los videojuegos analizados en el estado del arte, la propuesta del proyecto se aleja notablemente de los elevados niveles de dificultad que caracterizan a estos títulos. El enfoque del juego no estará centrado en ofrecer un desafío exigente, sino en hacer el título accesible a un mayor número de jugadores, permitiendo que todos puedan disfrutar de la experiencia sin barreras significativas. Además, mientras que los juegos mencionados se destacan por su narrativa, la propuesta de este proyecto se centrará aún más en un enfoque educativo a través de la narrativa. Este videojuego busca transmitir un mensaje profundo sobre la evolución emocional de un niño al atravesar las cinco fases del duelo tras la pérdida de un ser querido, ofreciendo una experiencia que, más allá del entretenimiento, tenga un propósito de concienciación y aprendizaje sobre temas de salud mental.

Por otro lado, tras analizar las diferentes alternativas, el motor gráfico seleccionado para el desarrollo del videojuego ha sido *Godot Engine*. Las razones se justifican con más detalle en el apartado 4.2 del capítulo posterior.

CAPÍTULO 3:

ANÁLISIS DEL CASO DE ESTUDIO DEL PROYECTO

A continuación, se presenta un análisis del producto y de los factores externos que pueden influir en este proyecto de software. Aunque este documento se enfoca principalmente en la primera iteración del proyecto, se incluirán menciones a algunas características que estarán relacionadas con el producto final.

3.1. Concepto del juego y sinopsis

Oniros es un videojuego en el que controlamos a un niño guerrero el cual debe atravesar las cinco tierras de un reino ubicado en sus sueños, donde cada una de estas representa una fase de la pérdida.

La historia de este juego comienza con el protagonista yéndose a dormir, el día después de que su mascota falleciera. Aprovechando este momento de debilidad, el guardián de los sueños renegado, Epiales, atrapa al chico en un sueño eterno. Para poder despertar, deberá afrontar su dolor y, con la ayuda de otros guardianes y dioses del sueño, derrotar a Epiales para evitar que siga atrapando humanos en sus pesadillas.

Mientras trata de escapar de este reino de sueños, deberá resolver acertijos y hacer frente a innumerables enemigos (conocidos generalmente como pesadillas) así como a los gobernantes de dichas tierras. El juego alcanza su fin una vez el protagonista ha atravesado los cinco reinos, representando así que ha conseguido superar todas las fases de la pérdida.

3.2. Características principales

Los principales pilares del juego son los siguientes:

1. Narrativa mixta de complejidad media: la narrativa principal de este juego trata temas psicológicos desde el punto de vista de un niño, junto con otra historia complementaria original relacionada con la mitología griega. La primera de las dos se cuenta usando la metodología de narrativa de mundo, utilizando los objetos, los coleccionables y el diseño de niveles del juego como mecanismo narrativo; por otra parte, la complementaria se cuenta a través de las interacciones entre los personajes y las escenas animadas.
2. Estrategia: el jugador deberá saber cómo gestionar sus recursos y sus habilidades, así como determinar los patrones de actuación de los enemigos antes de abordar un enfrentamiento. Esto se debe a que no contará durante toda la partida con la posibilidad de modificar la totalidad de sus equipamientos, por lo que deberá tomar una serie de decisiones antes de abandonar la zona habilitada para el reabastecimiento de equipo o el establecimiento de mejoras. También deberá determinar cómo abordar cada acción de exploración, puesto que los puntos de guardado son muy limitados.
3. No linealidad y *Backtracking*: aunque la historia del juego es lineal, la exploración no lo es, por lo que se le da la posibilidad al jugador de acceder a zonas opcionales en cada una de las zonas, así como regresar a ellas posteriormente para obtener mejoras, coleccionables o realizar misiones secundarias (Devuego, s.f.). Esta posibilidad de

exploración antes de seguir avanzando en la historia es lo que se denomina como *backtracking*.

4. Dinamismo y combate activo: el juego cuenta con muchos componentes de plataforma, por lo que el personaje estará prácticamente en constante movimiento. Además, los combates serán dinámicos (a diferencia de un sistema de combate por turnos), contando los enemigos con mecánicas diferentes entre ellos que el jugador deberá analizar.

Aunque se prevé que estos cuatro aspectos estén en la versión final del videojuego, esta primera iteración solo mostrará aproximadamente tres de ellos (no se mostrará aun nada de la narrativa), correspondiendo mayoritariamente a una demostración técnica que permita ver el correcto funcionamiento de las mecánicas y las posibilidades que ofrece el motor gráfico.

3.3. Género

Para que se entienda correctamente la clasificación del juego, se aporta a continuación una pequeña descripción de todos los géneros que abarca. Los géneros en cuestión son:

- Plataformas: de manera genérica, los juegos de plataformas (Caurin, 2019) se definen como videojuegos en los que el jugador debe hacer avanzar el personaje principal a través de una serie de plataformas suspendidas o móviles, mientras hace frente a una serie de enemigos, con el objetivo de llegar a una meta predefinida.
- *Metroidvania*: este género realmente es un caso especial o subgénero de los videojuegos de plataformas. Cuenta con las mismas mecánicas básicas, pero a diferencia del género mencionado anteriormente, este tipo de juego no se divide por niveles. Por norma general (Quesada, 2020), se caracteriza principalmente por ser un tipo de videojuego de plataformas que no es lineal. Dicho de otra manera, en estos juegos el usuario no avanza por las pantallas de manera unidireccional (como sucedería en Super Mario Bros), sino que se le plantea al jugador un mapa con una sucesión de pantallas que puede visitar de manera libre, adaptando a su libre albedrío el tiempo y la dedicación que desee ofrecer hasta llegar a la meta.
- *Souls-like*: este tipo de juegos (Andalucía Game, 2024) proceden realmente como un subgénero de los juegos de rol de acción o de acción y aventura, y se caracterizan principalmente por tener unos altos niveles de dificultad (ofreciendo a los jugadores una gran dualidad entre desafío y recompensa) y por tener una muy desarrollada narrativa de mundo, es decir: no priorizan tanto el contar la historia mediante los personajes o el propio guion del juego, sino que se enfocan en diseñar y plantear correctamente la ambientación del juego y la historia que subyace, siendo al final mucho más importante el marco en el que se ambienta la historia del juego que la propia historia del juego. Su mecánica más destacable es el combate contra "jefes finales", que no son más que enemigos de muy alto nivel del juego que se hallan al final de cada una de las zonas y que aportan un gran desafío para los jugadores, generalmente en batallas de 1 vs 1. El mayor exponente de este subgénero sería la saga Dark Souls, por la que recibe su nombre.

3.4. Propósito y público objetivo

El objetivo principal de este videojuego es el de ofrecer una historia emotiva y con un alto componente psicológico que permita representar una posible interpretación de las diferentes fases de la pérdida. Al mismo tiempo, se busca que el jugador no solo sienta una profunda inmersión en la obra, sino que sienta una jugabilidad dinámica y con un alto entretenimiento.

Esta obra está enfocada hacia todos los públicos, aunque por el nivel de dificultad establecido se recomienda mayoritariamente a jugadores más experimentados con el género *Metroidvania* o *Souls-like*. Su dificultad no será tan elevada como otros títulos del estilo para poder facilitar la accesibilidad a nuevos jugadores, pero también es cierto que puede generar mucho más nivel de frustración para estos precisamente por ser más difícil que la media de juegos.

3.5. Estilo visual

El estilo visual del juego está muy inspirado en el estilo de juegos como *Blasphemous* o los *The Last Faith* (ver ilustración 7), aunque el diseño del entorno tendrá unas texturas bastante simples en comparación con estos.

Ilustración 7. Imagen de gameplay de The Last Faith. Fuente: HobbyConsolas, 2023

Además, la paleta de colores no será extremadamente realista especialmente para el diseño de las pantallas, ya que se crearán las combinaciones en función de los colores que son generalmente asociados a cada una de las etapas de la pérdida (por ejemplo, en el caso de la negación, el entorno tomará unos colores más cercanos a los tonos morados,

grises y azules). No obstante, se aplicará un diseño más detallado y realista de cara a los personajes, para que queden bien diferenciados entre ellos.

Por el momento, la demostración técnica sobre la que versa este documento no contará con los diseños finales, puesto que actualmente el videojuego está siendo desarrollado por el autor del documento, que no posee los conocimientos adecuados de diseño artístico para videojuegos. Esto implica que los diseños observados en la demostración son extraídos de paquetes de recursos libres de derechos de autor y encontrados de manera gratuita en internet, los cuáles son mostrados y referenciados en el Anexo 2 de este documento.

3.6. Alcance del proyecto

El objetivo principal es desarrollar un juego con el menor número de errores posible y que permita transmitir una historia compleja e inmersiva de manera visual, al mismo tiempo que incita al jugador explorar todas las zonas del juego. Además, también se desea que el juego pueda ser ejecutable en dispositivos que cuenten con un sistema operativo Windows 10 o superior, un procesador Intel Core i5 o similar/superior, un mínimo de 4GB de RAM y 4GB de memoria disponible.

Como se comentaba anteriormente, la idea de partida de esta primera iteración es desarrollar una *demo* técnica que nos permita medir su rendimiento y valoración con *feedback* de los usuarios, analizando si las mecánicas establecidas convencen a los jugadores y si están bien diseñadas e implementadas. En total, las funcionalidades que se desean implementar en la versión final del videojuego son las siguientes:

1. **Movimiento del personaje**: Permite al jugador desplazarse en la dirección deseada (derecha o izquierda).
2. **Salto**: Habilita al jugador para realizar un salto.
3. **Doble salto**: Permite al jugador ejecutar un segundo salto en el aire.
4. **Ataque**: Permite al jugador atacar a los enemigos.
5. **Bloquear ataque**: Habilita al jugador para bloquear los ataques enemigos.
6. **Deslizamiento o "Dash"**: Permite al jugador moverse rápidamente en una dirección.
7. **Recibir daño**: Reduce la vida del personaje cuando es atacado por un enemigo.
8. **Muerte del personaje**: Elimina al personaje de la pantalla cuando su vida llega a cero.
9. **Movimiento de los enemigos**: Permite a los enemigos desplazarse por el entorno.
10. **Patrullaje de enemigos**: Define el comportamiento de los enemigos al patrullar, cambiando de dirección al detectar paredes o bordes sin suelo.
11. **Detección del jugador por enemigos**: Permite a los enemigos cambiar su dirección de desplazamiento al detectar al jugador.
12. **Ataque de enemigos**: Habilita a los enemigos para atacar al jugador cuando está en su área de ataque.
13. **Recibir daño por parte de los enemigos**: Reduce la vida del enemigo cuando es atacado por el jugador.
14. **Interacción con estatuas de guardado**: Permite al jugador guardar la partida y recuperar vida al interactuar con estatuas específicas.

15. **Consulta de estadísticas del personaje**: Permite al jugador visualizar las estadísticas de su personaje.
16. **Sistema de mejoras**: Permite mejorar los atributos del personaje en los puntos de guardado empleando los puntos obtenidos al derrotar enemigos, los cuales se verán también reflejados en el menú de estadísticas.
17. **Sistema de inventario y equipamientos**: Este sistema permite al jugador gestionar objetos recolectados durante el juego, como armas, armaduras y otros ítems. Los jugadores pueden equipar o desequipar estos objetos para mejorar las habilidades del personaje o para superar ciertos obstáculos.
18. **Sistema de viaje rápido**: Permite al jugador moverse rápidamente entre diferentes puntos del mapa que ya ha explorado, lo que facilita la navegación y el retorno a zonas previamente visitadas sin tener que recorrerlas nuevamente.
19. **Sistema de diálogos, personajes no jugables y misiones secundarias**: Este sistema gestiona las interacciones del jugador con personajes no jugables (PNJs). A través de diálogos, el jugador puede recibir misiones secundarias, obtener información importante o influir en el desarrollo de la historia.
20. **Zonas ocultas opcionales**: Se trata de áreas del juego que no son necesarias para completar la historia principal, pero que pueden contener recompensas especiales, como objetos raros, mejoras, o partes adicionales de la narrativa.
21. **Personaje acompañante**: Implementa un perro controlado por la inteligencia artificial que acompaña al jugador durante su aventura y que se considera el protagonista secundario del juego. Este personaje puede ayudar en combate, resolver rompecabezas o proporcionar apoyo narrativo.
22. **Cinemáticas y elementos de narrativa**: Incluye secuencias de video o animaciones que desarrollan la historia, proporcionando contexto y profundidad emocional. Estos elementos se utilizan para avanzar la trama y conectar las diferentes partes del juego.
23. **Coleccionables**: Objetos que los jugadores pueden recolectar a lo largo del juego, generalmente ocultos o difíciles de conseguir. Estos pueden desbloquear contenido adicional, como mejoras, trajes, o simplemente servir como logros.
24. **Barra de magia y ataques especiales mágicos**: Introduce una barra de magia que se consume al usar habilidades o ataques especiales. Estos ataques son más poderosos que los normales y pueden ser esenciales para superar ciertos enemigos o desafíos.
25. **Rompecabezas complejos que impliquen el desplazamiento entre zonas o disponer de algún objeto en el inventario**: Son desafíos que requieren que el jugador utilice su ingenio para resolver problemas que involucran múltiples áreas del juego. Algunos rompecabezas pueden necesitar objetos específicos del inventario o la activación de mecanismos en diferentes partes del mapa.
26. **Diseño de niveles complejo estilo *metroidvania***: Crea un mundo interconectado donde el acceso a nuevas áreas depende de habilidades o herramientas adquiridas, incentivando la exploración y la rejugabilidad.

De todas las funcionalidades especificadas, y teniendo en cuenta que se quiere comprobar que las funcionalidades básicas del juego funcionan correctamente, en la versión correspondiente a la demo técnica de este documento se van a implementar las 15 primeras de la lista, que son consideradas las más básicas y, por ende, prioritarias para la experiencia de juego.

CAPÍTULO 4:

MARCO DE TRABAJO Y HERRAMIENTAS DE DESARROLLO DEL PROYECTO

4.1. Marco de desarrollo del proyecto

En una primera instancia este proyecto iba a ser realizado en conjunto con otras personas que se encargarían de las diferentes partes comunes a un videojuego (diseños de artes conceptuales, diseño de personajes y paisajes, composición musical y diseño de sonido) que no tienen que ver con la programación. No obstante, por problemas de compatibilidad de horarios y fechas, el equipo inicial quedó disuelto, quedando finalmente en un videojuego desarrollado únicamente por el autor del documento.

Esto hizo que se tuviera que cambiar la metodología de trabajo planificada inicialmente, pasando de una metodología SCRUM a una metodología clásica basada en el modelo incremental explicado en la asignatura de Ingeniería del Software, donde se seleccionaron una serie de usuarios cercanos al autor del documento (y con experiencia jugando a videojuegos) que servirían como método de evaluación de los diferentes incrementos realizados en cada una de las iteraciones del proyecto.

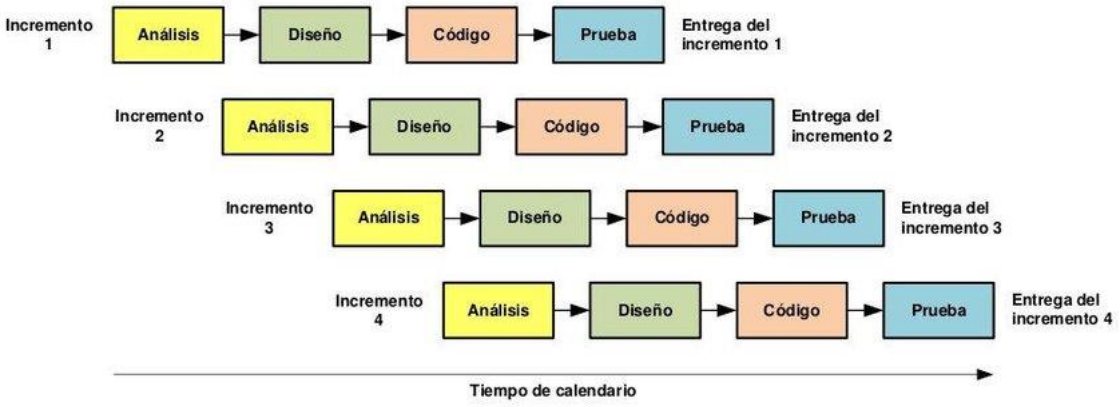


Ilustración 8. Esquema de modelo de desarrollo de software incremental. Fuente: ResearchGate, s.f.

Tras recopilar los requisitos del alcance y teniendo en cuenta el método de desarrollo empleado, se procedió a planificar el proyecto de manera estructurada, dividiéndolo en cinco incrementos hasta alcanzar la versión del producto tras la primera iteración. Cada incremento representó un conjunto de funcionalidades y mejoras, implementadas de forma progresiva para garantizar un desarrollo ordenado y eficaz del videojuego. Esta planificación permitió mantener un control sobre el avance del proyecto, asegurando que cada etapa cumpliera con los objetivos establecidos antes de pasar al siguiente incremento. La tabla presentada a continuación muestra la planificación detallada de estos incrementos.

Nº DE INCREMENTO	**TAREAS A REALIZAR**
Incremento 1	• Formación en el entorno de desarrollo y lenguaje. • Obtención de recursos como plantillas para el diseño de personajes y el mapa, o la música de las escenas. • Configuración del proyecto y el mapa de entrada (controles) • Creación de una escena de prueba para probar el funcionamiento de las escenas instanciables. • **Creación de un script general que sirva para la persistencia de los datos y la interacción entre escenas.**
Incremento 2	• Creación del personaje jugable. • Selección y aplicación del diseño para el personaje. • Configuración de las animaciones y asociación al mapa de entrada. • Creación del HUD (puntuación, barra de vida y feedback de guardado). • Implementación de las funcionalidades del personaje (funcionalidades de 1 a 8).
Incremento 3	• Creación de los tres tipos de enemigo instanciables. • Selección y aplicación de diseños para los enemigos. • Configuración de las animaciones de los enemigos. • Diseño de las interfaces de pausa, visualización de estadísticas y menú principal. • Implementación de las funcionalidades de enemigos normales (funcionalidades de 9 a 13) e interfaces (15)
Incremento 4	• Diseño y creación de las escenas jugables (zonas del mapa). • Diseño y creación de la escena de combate final. • Diseño y creación de la escena de la estatua, empleada para el guardado. • Diseño y creación de fin de la demo. • Implementación del sistema de guardado (func. 14), de las funcionalidades del jefe final (func. 9 a 13) y del fichero general (persistencia y transición entre escenas conservando datos)
Incremento 5	• Pruebas de las escenas instanciables y no instanciables. • Corrección de errores de la primera fase de pruebas. • Adición de música a las escenas. • Creación de animaciones con efectos de difuminado para suavizar las transiciones. • Exportación del proyecto a archivo ejecutable para Windows y realización de pruebas de usuario

Como se puede observar, la planificación del proyecto se divide en cinco incrementos, donde cada uno se enfoca en la implementación progresiva de diferentes funcionalidades esenciales del videojuego. Sin embargo, es importante destacar que los incrementos 1 y 5 presentan particularidades que los diferencian del ciclo en cascada tradicional. El primer incremento está dedicado principalmente a la preparación del entorno y los recursos necesarios para el desarrollo, mientras que el quinto incremento se centra en la realización de pruebas exhaustivas y la incorporación de elementos adicionales como la música y las transiciones, con el objetivo de pulir y completar el producto final. Estas características hacen que ambos incrementos tengan un enfoque más preparatorio y de validación en comparación con los otros, que siguen un proceso más lineal de desarrollo.

4.2. Herramientas de desarrollo del proyecto

A continuación, se presentan las plataformas y herramientas empleadas para el desarrollo de este videojuego:

- **Godot Engine**. De todos los motores especificados en el estado del arte, se ha escogido Godot Engine por tres razones: en primer lugar, aunque no existen juegos de gran categoría asociados a este motor, si es muy compatible con la creación de videojuegos en 2D, por lo que eso de partida lo convierte en un motor adecuado para la propuesta; en segundo lugar, es muy intuitivo de manejar, gracias a su editor visual, formado por un sistema *drag and drop* similar al de aplicaciones como *Scene Builder* (usado en con JavaFX) que, no solo facilita la composición de las escenas, sino que además permite interconectar la capa de presentación con los *scripts* del videojuego por medio de manejadores de eventos; y en tercer lugar por su lenguaje integrado *GDScript*, que es un lenguaje de *scripting* orientado a objetos cuya adaptación ha sido más sencilla (reduciendo así la curva de aprendizaje) dadas sus similitudes sintácticas con Python o JavaScript, lenguaje bastante familiar para el autor del documento.
- **GitHub y GitHub Desktop** (GitHub, s.f.): es una plataforma de desarrollo colaborativo que se basa en Git, un sistema de control de versiones distribuido. GitHub permite a los desarrolladores almacenar, gestionar y compartir su código, facilitando la colaboración en proyectos de software. En el contexto de este proyecto se ha empleado como herramienta de control de versiones y creación de copias de seguridad, las cuales han sido bastante útiles en ciertas circunstancias del proyecto donde por errores del código se ha tenido que regresar a versiones anteriores. Para poder agilizar este control de versiones se ha empleado la GitHub Desktop, una aplicación de escritorio de GitHub que permite realizar las operaciones con el repositorio creado en remoto en la web de GitHub sin necesidad de memorizar comandos de Git, sino que se empleaba en todo momento una GUI (o Interfaz Gráfica de Usuario).
- **Canva** (Fernández, 2023): es una plataforma de diseño gráfico en línea que permite a los usuarios crear una amplia variedad de contenidos visuales, como presentaciones, carteles, infografías y publicaciones para redes sociales. Ofrece una interfaz intuitiva con herramientas de arrastrar y soltar, además de una amplia biblioteca de plantillas, imágenes y elementos gráficos, lo que facilita el diseño tanto para profesionales como para personas sin experiencia en diseño gráfico. En

este caso se ha empleado principalmente para la maquetación final de los prototipos de mapas del videojuego (ver Ilustración 9), para poder tener una visión completa de las zonas jugables.

Ilustración 9. Fragmento de un prototipo de mapa del juego descartado. Fuente: Elaboración propia, 2024

- **Udemy** (Udemy, s.f.): es una plataforma de aprendizaje en línea que ofrece una amplia gama de cursos en diversas áreas como tecnología, negocios, arte y desarrollo personal. Los cursos son creados por instructores de todo el mundo y están diseñados para ayudar a los estudiantes a adquirir nuevas habilidades o mejorar las existentes, a su propio ritmo. Udemy facilita el acceso a la educación para millones de personas, con opciones tanto gratuitas como de pago. Ha sido la plataforma empleada en el proceso de formación en el motor gráfico Godot y su lenguaje integrado GDScript.
- **Itch.io** (Itch.io, s.f.): es una plataforma en línea que permite a desarrolladores independientes publicar, distribuir y vender sus juegos y proyectos digitales. Fundada con el objetivo de dar visibilidad y soporte a los creadores indie, Itch.io ofrece un espacio donde estos pueden establecer sus propios precios y términos de distribución, conectando directamente con su audiencia. Además, la plataforma fomenta la creatividad y la innovación en el ámbito de los videojuegos independientes. Esta plataforma ha sido empleada en el trabajo para poder extraer los recursos empleados para el diseño de los niveles, así como los estilos tanto de las fuentes como de las propias interfaces (ver Anexo 2).

CAPÍTULO 5:

ANÁLISIS Y ESPECIFICACIÓN DE REQUISITOS

5.1. Requisitos funcionales

En este apartado se presentarán los requisitos funcionales del juego, los cuales definirán las características esenciales que permitirán la interacción del jugador con el sistema. Se abordará el flujo de juego, describiendo los pasos generales que el jugador seguirá en cada partida, los casos de uso que ilustrarán las diferentes acciones posibles dentro del juego, y los controles, que detallarán las opciones disponibles para la manipulación y navegación en el entorno del juego.

5.1.1. Flujo de juego

El diagrama presentado (ver Ilustración 10) muestra el flujo de juego diseñado para el videojuego, ilustrando las etapas y decisiones clave que el jugador enfrentará durante una sesión de juego. Desde el inicio de la partida hasta la posible conclusión de la sesión, el flujo abarca la creación y selección de espacios de guardado, la reanudación de la partida desde puntos de control, la actualización de estadísticas, y las decisiones que debe tomar el jugador en caso de morir o querer salir del juego.

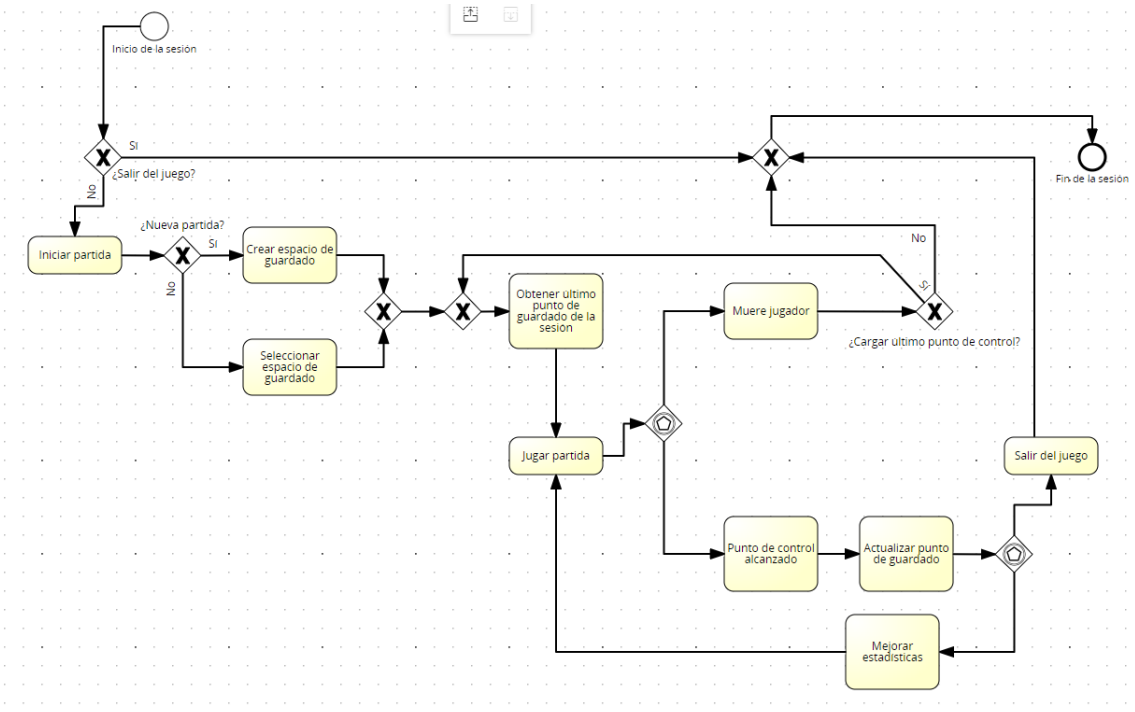


Ilustración 10. Flujo de juego general de cada sesión. Fuente: elaboración propia, 2024

5.1.2. Casos de uso

Los casos de uso se dividen principalmente en 4 bloques, los cuáles se presentan a continuación:

- PERSONAJE JUGABLE

REFERENCIA	CU-01
NOMBRE	Movimiento
DESCRIPCIÓN	Permite al jugador desplazarse. 1. El jugador pulsa el botón de dirección de movimiento. 2. El sistema comprueba si la dirección es derecha o izquierda. 3. El sistema aplica al personaje una velocidad al personaje hacia la dirección detectada.
ACTOR	Usuario

REFERENCIA	CU-02
NOMBRE	Salto
DESCRIPCIÓN	Permite al jugador realizar un salto. 1. El jugador pulsa el botón de salto. 2. El sistema comprueba si el personaje está en el suelo. 3. El sistema aplica al personaje una velocidad hacia arriba.
ACTOR	Usuario

REFERENCIA	CU-03
NOMBRE	Doble salto
DESCRIPCIÓN	Permite al jugador realizar un doble salto. 1. El jugador pulsa el botón de salto. 2. El sistema comprueba si el personaje no está en el suelo. 3. El sistema comprueba si solo se ha realizado un salto. 4. El sistema aplica al personaje una velocidad hacia arriba.
ACTOR	Usuario

REFERENCIA	CU-04
NOMBRE	Ataque
DESCRIPCIÓN	Permite al jugador atacar a los enemigos. 1. El jugador pulsa el botón de ataque. 2. El sistema comprueba si el personaje está en el suelo. 3. El sistema ejecuta la acción de ataque y genera un área de colisión para dañar a los posibles enemigos.
ACTOR	Usuario

REFERENCIA	CU-05
NOMBRE	Bloquear ataque
DESCRIPCIÓN	Permite al jugador bloquear los ataques enemigos. 1. El jugador pulsa el botón de bloqueo. 2. El sistema comprueba si el personaje está en el suelo. 3. El sistema ejecuta la acción de bloqueo y genera un área de colisión para anular los posibles ataques enemigos.
ACTOR	Usuario

REFERENCIA	CU-06
NOMBRE	*Dash*
DESCRIPCIÓN	Permite al jugador dar un impulso de velocidad en una dirección. 1. El jugador pulsa el botón asociado al *dash*. 2. El sistema comprueba que el personaje esté en movimiento. 3. El sistema aplica momentáneamente un incremento de la velocidad.
ACTOR	Usuario

REFERENCIA	CU-07
NOMBRE	Recibir daño
DESCRIPCIÓN	Reduce la vida del personaje jugable. 1. Un enemigo manejado por el sistema realiza un ataque. 2. El sistema comprueba que el jugador esté en el área de ataque enemiga. 3. El sistema reduce en uno la barra de vida del jugador.
ACTOR	-

REFERENCIA	CU-08
NOMBRE	Muerte
DESCRIPCIÓN	Elimina al personaje de la pantalla. 1. Tras aplicar daño al jugador el sistema comprueba si los puntos de vida son iguales a cero. 2. El sistema elimina al jugador de la pantalla y muestra el mensaje “Has muerto”.
ACTOR	-

En la ilustración 11, podemos ver un diagrama de transición entre los diferentes estados en los que puede encontrarse el personaje, determinando que acciones puede llevar a cabo en función del estado en el que se encuentre.

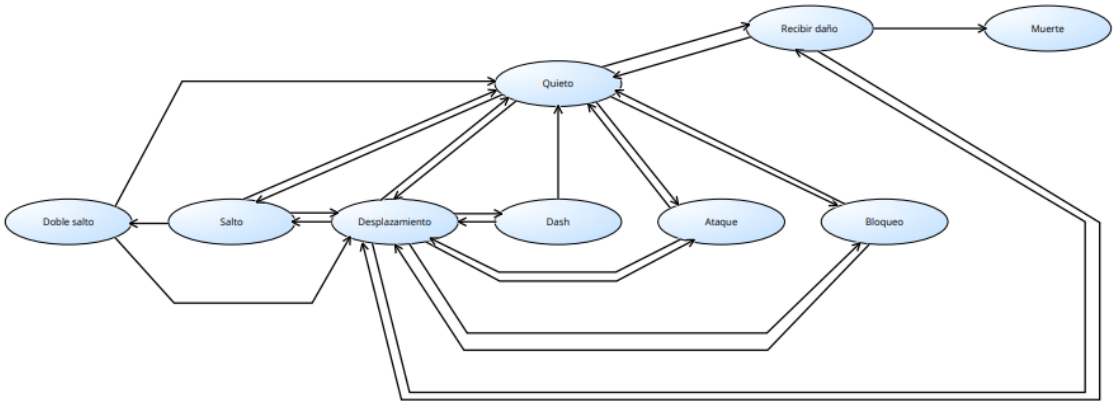


Ilustración 11. Diagrama de estados del personaje jugable. Fuente: Elaboración propia, 2024

- ENEMIGOS

REFERENCIA	CU-09
NOMBRE	Movimiento
DESCRIPCIÓN	Permite al enemigo desplazarse. 1. El sistema determina un sentido de desplazamiento en función de lo que determinen los casos de uso 10. 11 y 12. 2. El sistema aplica una velocidad en el sentido establecido.
ACTOR	-

REFERENCIA	CU-10
NOMBRE	Patrulla 1
DESCRIPCIÓN	Permite al enemigo establecer el sentido de desplazamiento en función de si detecta o no una pared. 1. Mientras está en movimiento, el sistema comprueba constantemente si el enemigo está próximo a una pared. 2. Si se detecta una pared, el sistema aplica la misma velocidad al enemigo, pero cambiando el sentido de este
ACTOR	-

REFERENCIA	CU-11
NOMBRE	Detección de jugador
DESCRIPCIÓN	Permite al enemigo establecer el sentido de desplazamiento en función de si detecta o no al jugador. 1. Mientras está en movimiento, el sistema comprueba constantemente si el enemigo está próximo a un jugador. 2. Si se da el caso de que lo detecta en un sentido diferente al del desplazamiento, el sistema cambiará el sentido de desplazamiento del enemigo. En caso contrario mantendrá el sentido.
ACTOR	-

REFERENCIA	CU-12
NOMBRE	Patrulla 2
DESCRIPCIÓN	Permite al enemigo establecer la dirección de desplazamiento en función de si detecta o no un borde (ausencia de suelo). 1. Mientras está en movimiento, el sistema comprueba constantemente si el enemigo está próximo a un espacio sin suelo. 2. Si detecta un espacio sin suelo, el sistema cambia el sentido de desplazamiento para evitar la caída.
ACTOR	-

REFERENCIA	CU-13
NOMBRE	Ataque
DESCRIPCIÓN	Permite al enemigo atacar al jugador. 1. El sistema comprueba que el jugador se encuentre dentro del área de ataque del enemigo. 2. El sistema ejecuta el ataque.
ACTOR	-

REFERENCIA	CU-14
NOMBRE	Recibir daño
DESCRIPCIÓN	Reduce la vida del enemigo 1. El sistema comprueba si el enemigo se encuentra dentro del área de ataque del jugador. 2. Si es así, el sistema reduce la vida del enemigo en la cantidad de daño establecida (la cual depende del nivel de dificultad del enemigo). 3. Si además es la primera vez que el enemigo recibe daño, se mostrará encima de este su barra de vida, que mientras tiene la vida al completo permanece oculta.
ACTOR	Usuario

Algunas anotaciones convenientes dentro de la especificación de estos casos de uso:

- Los casos de uso 10, 11 y 12 no son excluyentes entre sí, por lo que pueden existir enemigos que tengan o uno solo de estos sistemas de detección, o que cuenten con los tres al mismo tiempo.
- No se ha especificado el caso de uso de muerte del enemigo por dos razones: en primer lugar, porque funciona como el caso de uso 8, pero aplicando la lógica a un enemigo manejado por el sistema; y, en segundo lugar, porque existe un tipo de enemigo que no desaparece de la pantalla tras eliminarlo, ya que hay que derribarlo varias veces, aunque este caso se explicará posteriormente en el capítulo 7, sobre implementación de cada uno de los componentes esenciales.

En la ilustración 12, podemos ver un diagrama de transición entre los diferentes estados en los que puede encontrarse un enemigo, determinando que acciones puede llevar a cabo en función del estado en el que se encuentre.

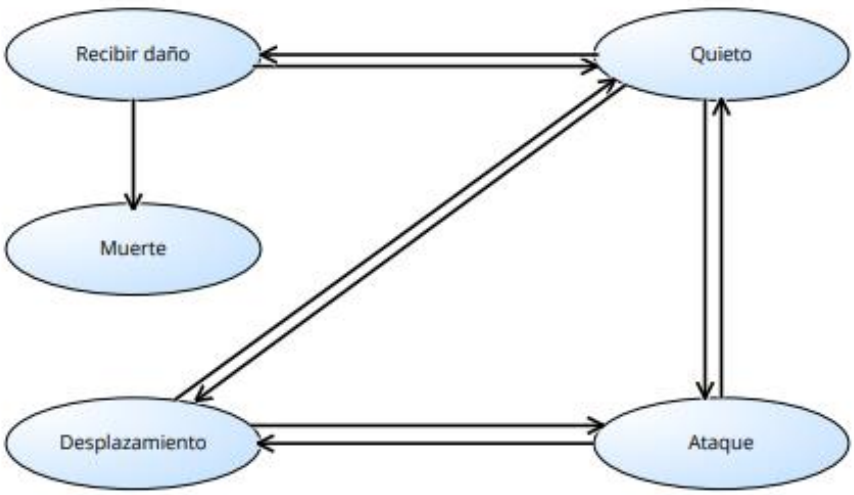


Ilustración 12. Diagrama de estados de los enemigos. Fuente: Elaboración propia, 2024

- SISTEMA DE GUARDADO

REFERENCIA	CU-15
NOMBRE	Tocar estatua
DESCRIPCIÓN	Permite al jugador guardar partida y recuperar vida. 1. El sistema comprueba si el jugador se encuentra dentro del área de interacción de la estatua. Si es así, muestra encima de esta un indicador de interacción. 2. El jugador pulsa el botón de interacción. 3. El sistema guarda la partida y restablece al completo los puntos de vida del jugador. 4. El sistema establece la posición de la estatua como punto de reaparición del jugador en caso de muerte.
ACTOR	Usuario

- SISTEMA DE ESTADÍSTICAS DE PERSONAJE

REFERENCIA	CU-16
NOMBRE	Consultar estadísticas
DESCRIPCIÓN	Permite al jugador consultar sus estadísticas. 1. El jugador abre el menú de pausa. 2. El jugador navega hasta la ventana de atributos de estado.
ACTOR	Usuario

A modo de resumen, podemos encontrar en la ilustración 13 el diagrama de casos de uso que muestra el conjunto de interacciones del jugador con el sistema formado por el videojuego.

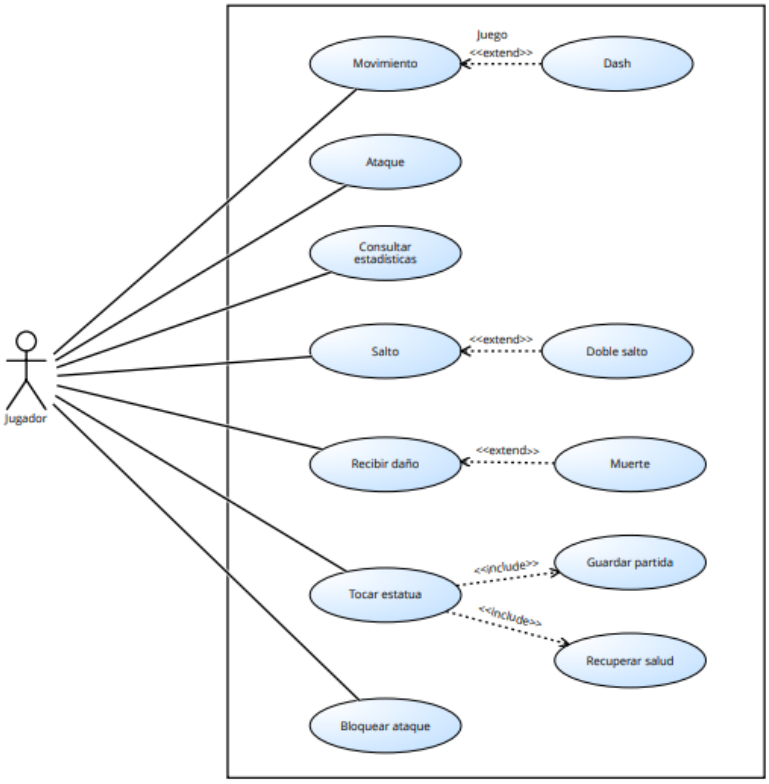


Ilustración 13. Diagrama de casos de uso. Fuente: Elaboración propia, 2024

5.1.3. Controles

A continuación, se muestran los controles del personaje principal para el jugador. Por el momento, la entrada solo está configurada para teclado y ratón y mandos de consola (*PlayStation 4/5, Xbox Series X y Nintendo Switch)*.

CONTROLES		
ACCIÓN	**ENTRADA MANDO (Ver Ilustración 15)**	**ENTRADA TECLADO/RATÓN**
Mover arriba	Joystick 3 arriba	Tecla W
Mover abajo	Joystick 3 abajo	Tecla A
Mover izquierda	Joystick 3 izquierda	Tecla S
Mover derecha	Joystick 3 derecha	Tecla D
Salto	Botón posición A	Tecla espacio
Doble salto	Salto x2	Salto x2
Deslizar o *dash*	Botón 9	Tecla X
Bloqueo	Botón 10	Botón derecho ratón
Ataque	Botón posición X	Botón izquierdo ratón
Interactuar	Cruz 2 arriba	Tecla E
Pausar partida	Botón 4	Tecla escape
Seleccionar (en un menú)	Joystick 3	Desplazamiento del ratón
Volver atrás	Botón posición B	Tecla escape

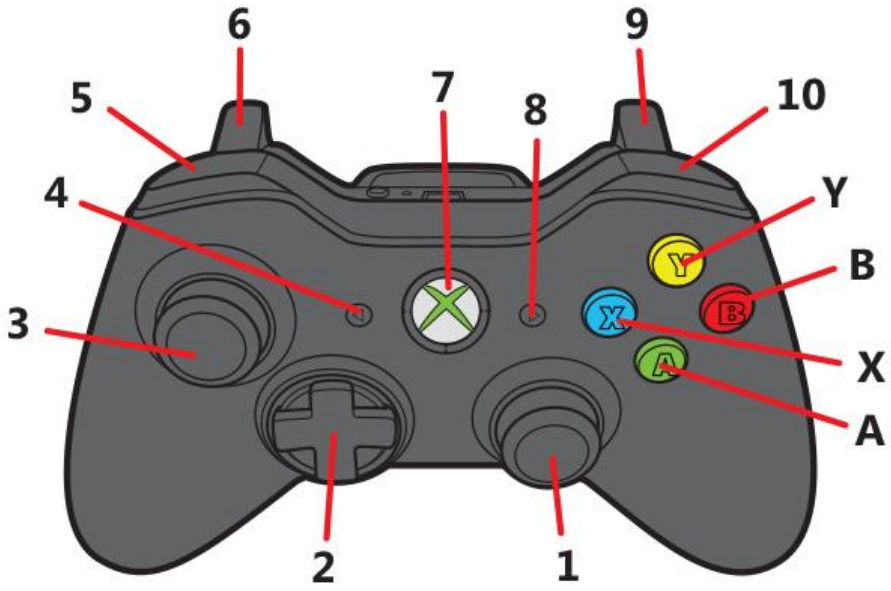


Ilustración 14. Ejemplo de mapa de entrada de mando. Fuente: Microsoft, s.f.

5.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que definen las características y restricciones del sistema que no están directamente relacionadas con funcionalidades específicas, pero que son esenciales para garantizar la calidad, rendimiento, seguridad y usabilidad del videojuego. En este apartado, se detallarán los aspectos clave que asegurarán que el sistema cumpla con los estándares esperados de eficiencia, mantenimiento y experiencia del usuario.

REQUISITOS NO FUNCIONALES	
REFERENCIA	**DESCRIPCIÓN**
RNF-01	El juego debe realizar su carga inicial en menos de 30 segundos y las escenas jugables deben cargar en menos de 10 segundos.
RNF-02	El videojuego no debe tener problemas de colisión en ninguna de las escenas.
RNF-03	El videojuego será ejecutable en sistemas con los siguientes requisitos mínimos: • SO: Windows 10 • Procesador: Intel Core i5 o AMD Ryzen 5 • Memoria: 4 GB de RAM • Almacenamiento: 4 GB
RNF-04	Los controles deben ser orientativos para cada tipo de dispositivo de entrada.
RNF-05	Los menús de pausa deben cargar de forma instantánea.

CAPÍTULO 6:

DISEÑO Y ARQUITECTURA DEL VIDEOJUEGO

6.1. Arquitecturas, patrones y tipos de programación empleados en el videojuego

En este apartado se abordarán los aspectos relacionados con la arquitectura, los patrones de diseño y los tipos de programación empleados en el desarrollo del videojuego. Aunque el proyecto no sigue una arquitectura concreta de manera estricta, su estructura está fuertemente influenciada por dos factores clave: el diseño de bucles de juego, que garantiza una experiencia fluida y coherente, y define la lógica del videojuego; y el patrón de arquitectura MVC (Modelo-Vista-Controlador), que facilita la organización y gestión del flujo de datos y la interacción entre los componentes del juego. Para manejar eficazmente todos los elementos de la partida y gestionar la integración de estos enfoques, se utilizará una combinación de programación orientada a eventos y a objetos, lo que permitirá un desarrollo modular y flexible del videojuego.

6.1.1. Diseño de Bucles de Juego

Un bucle de juego (Royhul, 2023), también conocido como *game core loop*, es la serie de acciones repetitivas que un jugador realiza durante el transcurso de un videojuego. Este ciclo constituye la esencia de la jugabilidad y define cómo interactúan los jugadores con el juego de manera continua. Los elementos fundamentales de un bucle de juego incluyen actividades básicas como la recolección de recursos, el combate, la exploración o la mejora de habilidades, que se repiten y se expanden a lo largo de la partida, ofreciendo a los jugadores un sentido de progreso y recompensa. La importancia del bucle de juego radica en su capacidad para mantener a los jugadores involucrados y comprometidos, asegurando que cada acción realizada se sienta significativa y contribuya al avance en el juego.

Además, determinar el bucle de juego del videojuego es de vital importancia, pues es lo que ayudará a determinar que acciones se realizan en el procesamiento de físicas del motor de videojuego y con qué frecuencia se hará, así como determinará cuál es la lógica final del videojuego. Un ejemplo asociado a la demo técnica sería el control de puntos de vida (ver Ilustración 15), donde en cada instante de tiempo (o iteración del bucle de juego), el motor de físicas está comprobando el número de puntos de vida restantes del jugador, estando pendiente de reducirlo en caso de que este reciba daño o terminando la partida en caso de que el jugador llegue a cero puntos de vida.

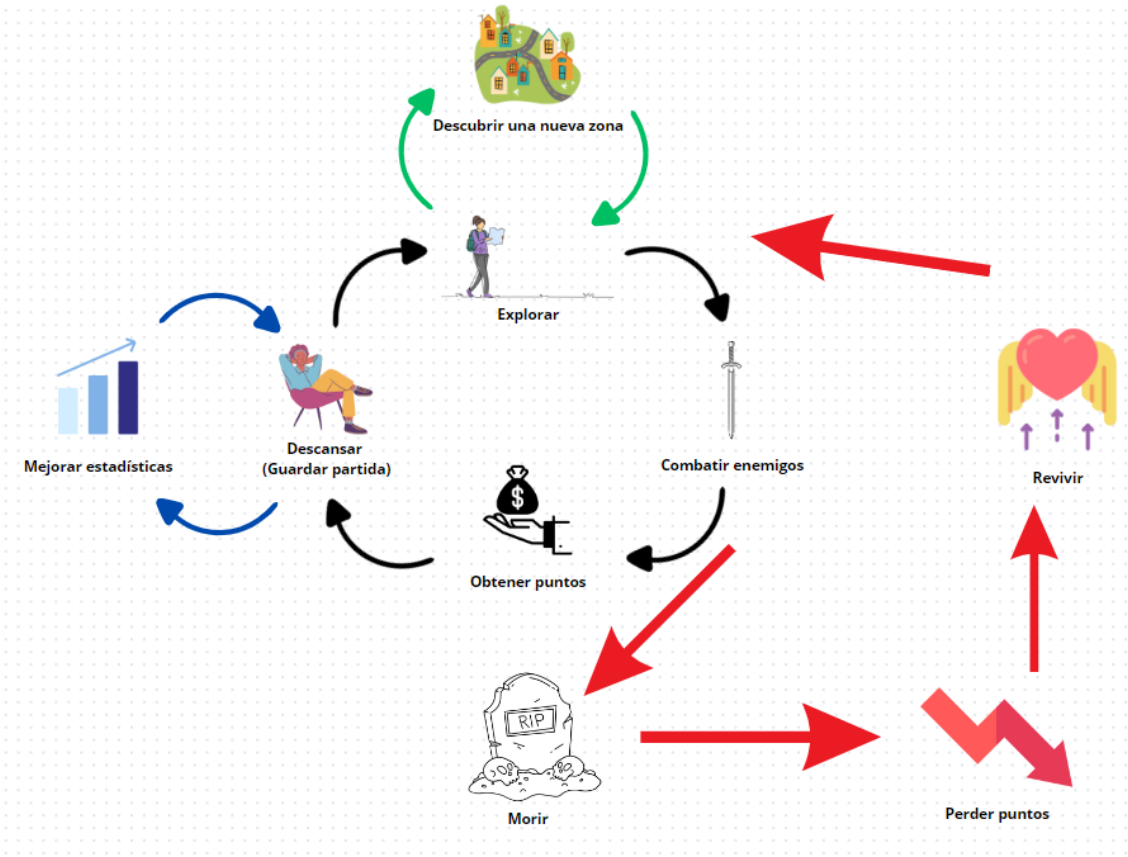


Ilustración 15. Bucle de juego de Oniros. Fuente: Elaboración propia, 2024

Este patrón de diseño ha influenciado notablemente tanto la estructuración del código como el flujo de juego que se podía vislumbrar en el capítulo anterior.

6.1.2. Patrón Modelo-Vista-Controlador

El patrón Modelo-Vista-Controlador (MVC) es un modelo de diseño de software que separa la lógica de la aplicación en tres componentes interrelacionados (Hernández, 2021): el Modelo, la Vista y el Controlador. El Modelo gestiona los datos y la lógica del negocio, la Vista se encarga de la representación visual de la información, y el Controlador actúa como un intermediario, gestionando las entradas del usuario y actualizando el Modelo y la Vista en consecuencia. Este patrón de diseño es fundamental para mantener un código organizado y facilitar la escalabilidad y el mantenimiento de aplicaciones complejas, al permitir que cada componente se desarrolle, pruebe y modifique de forma independiente. Dentro del contexto del proyecto, estos tres componentes pueden verse en el motor gráfico *Godot Engine* de la siguiente manera:

- **Modelo**: Aunque aún no corresponde elaborar demasiado acerca del guardado de datos, pues en la primera iteración no existe mucha información persistente, sí que es conveniente hablar sobre como este motor crea el modelo de datos. Godot (Godot Engine, s.f.) permite almacenar información de manera serializada en formato JSON para facilitar la persistencia de datos en los juegos. Este proceso implica convertir la información del juego, como el estado de los objetos o la configuración del jugador, en un formato JSON que se puede guardar fácilmente en archivos (ver Ilustración 16). La

serialización a JSON en Godot es útil porque este formato es ampliamente compatible y permite que los datos sean leídos y modificados fácilmente, lo que facilita tanto la recuperación como la actualización de la información guardada. Para implementar esto, Godot proporciona métodos integrados que convierten los datos a JSON y permiten su escritura en archivos, así como su posterior lectura y deserialización cuando se necesita restaurar el estado del juego. Por lo tanto y, a modo de resumen, el modelo de datos empleado no está formado por un tipo concreto de base de datos, sino que es un directorio formado por un conjunto de ficheros con datos de la partida de manera serializada.

GDScript C#

```gdscript
func save():
    var save_dict = {
        "filename" : get_scene_file_path(),
        "parent" : get_parent().get_path(),
        "pos_x" : position.x, # Vector2 is not supported by JSON
        "pos_y" : position.y,
        "attack" : attack,
        "defense" : defense,
        "current_health" : current_health,
        "max_health" : max_health,
        "damage" : damage,
        "regen" : regen,
        "experience" : experience,
        "tnl" : tnl,
        "level" : level,
        "attack_growth" : attack_growth,
        "defense_growth" : defense_growth,
        "health_growth" : health_growth,
        "is_alive" : is_alive,
        "last_attack" : last_attack
    }
    return save_dict
```

Ilustración 16. Ejemplo de diccionario para el guardado de datos. Fuente: Godot Engine, s.f.

- **Vista**: En el caso de Godot, la vista viene dada por el editor de nodos. Si observamos la ilustración 17, vemos dos elementos clave: en primer lugar, en la parte izquierda, se puede observar la estructura jerárquica de nodos en la escena, donde cada nodo representa un elemento con funcionalidades específicas. Esta jerarquía facilita la organización y gestión de los diferentes componentes del juego, permitiendo que cada nodo se comporte de manera independiente o en conjunto con otros; y, por último, en la parte derecha, encontramos el propio editor de la escena, que funciona de manera similar a un sistema *drag & drop*, comparable al de *Scene Builder,* mencionado anteriormente en este documento. Esta herramienta intuitiva permite a los desarrolladores diseñar y ajustar visualmente la disposición de los elementos en la escena, simplificando así el proceso de creación y modificación del entorno del juego.

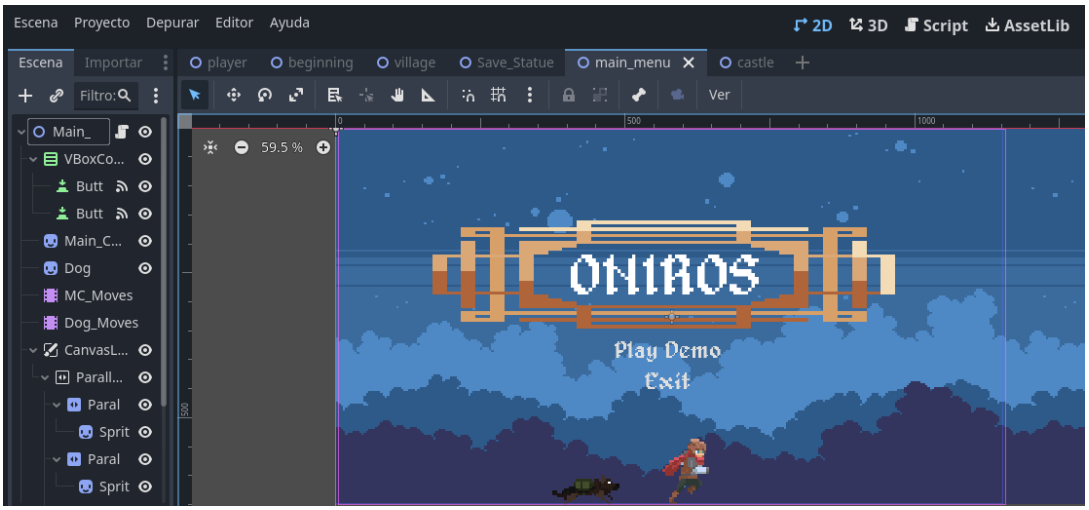


Ilustración 17. Estructura de nodos y editor de escenas de Godot. Fuente: Elaboración propia, 2024

- **Controlador**: En cuanto al controlador dentro del patrón MVC, Godot proporciona varias herramientas clave, visibles en la Ilustración 18: A la izquierda, se encuentra el listado de scripts existentes, que permite gestionar y seleccionar los diferentes scripts asociados a los nodos del proyecto; En el centro, se puede ver el editor de código integrado, donde los desarrolladores pueden escribir y modificar el comportamiento de los nodos mediante código; y a la derecha, se muestra el sistema de gestión de señales, un componente esencial en Godot que permite a cada nodo emitir señales o *listeners* en respuesta a acciones específicas. Estas señales pueden ser manejadas en los scripts seleccionados, funcionando como eventos que permiten un control dinámico y reactivo del juego.

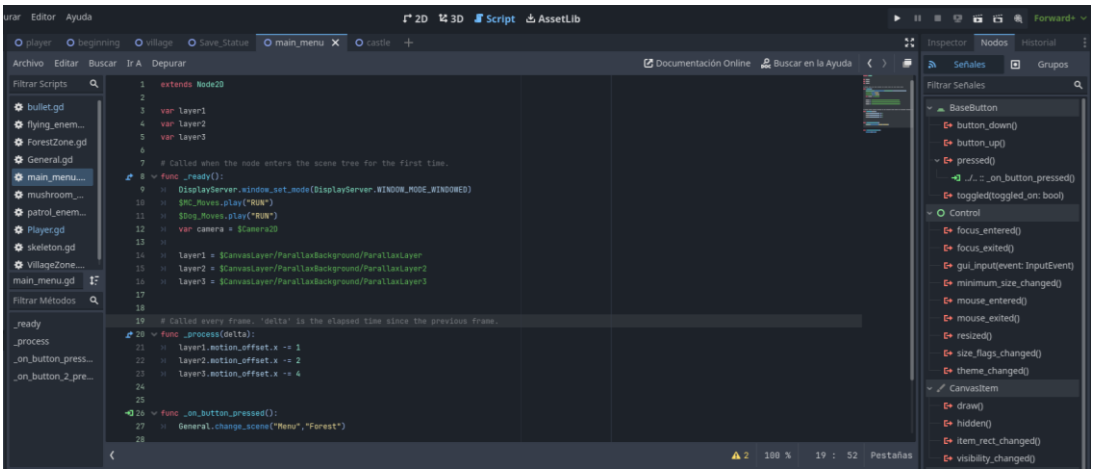


Ilustración 18. Editor de scripts y herramienta de asignación de listeners de Godot. Fuente: Elaboración propia, 2024

6.1.3. Programación Orientada a Eventos y Objetos

La programación orientada a objetos (POO) es un paradigma de programación que organiza el software en objetos, los cuales son instancias de clases (Universidad Europea, 2022). Cada objeto contiene datos en forma de atributos (propiedades) y comportamiento en forma de métodos (funciones). Este modelo permite la manipulación directa de los objetos, facilitando el desarrollo de software más modular, reutilizable y fácil de mantener. Dentro del contexto del proyecto, la programación orientada a objetos es útil para manejar nodos como *AnimationPlayer* (que permite crear y ejecutar diferentes animaciones en función de la acción realizada) o *Camera2D* (que permite establecer una cámara que siga al jugador, además de poder configurarse los límites de seguimiento en función de la escena en la que se encuentre), además de ser esencial para instanciar escenas como los enemigos (como se podrá observar en el siguiente capítulo).

Por otro lado, tenemos la programación orientada a eventos (Martínez, 2024), que es un paradigma de desarrollo de software en el que el flujo de la aplicación se determina mediante eventos, como acciones del usuario (clics, teclas presionadas) o mensajes de otros programas. En este enfoque, las aplicaciones están diseñadas para responder a una serie de eventos en lugar de seguir un flujo de control predefinido. Este modelo es ampliamente utilizado en interfaces gráficas de usuario (GUI), donde las interacciones del usuario desencadenan eventos que ejecutan funciones o métodos específicos. Dentro del contexto del proyecto, la programación orientada a eventos es esencial para aspectos tales como la gestión de la E/S (es decir, la interacción del usuario mediante mando o teclado) o la gestión de señales (como por ejemplo la superposición de dos objetos de colisión en pantalla).

6.2. Diseño y prototipado de interfaces

A continuación, se presentan las principales interfaces presentes en esta demo, incluyendo su boceto y su diseño final:

MENÚ PRINCIPAL: Dado que se trata de una versión de demostración, el menú se ha simplificado para incluir únicamente dos opciones: "Play Demo" y "Exit" (ver Ilustración 19). A diferencia de un juego completo, no se ofrecen opciones para crear una nueva partida o cargar una partida guardada. En esta demo, al seleccionar "Play Demo", los jugadores comenzarán la demo desde el principio o continuarán desde el último punto de guardado automáticamente. El diseño visual incluye un efecto de paralaje en el fondo, mientras que el personaje principal y su perro aparecen en la parte inferior de la pantalla, ambos en una animación de carrera.

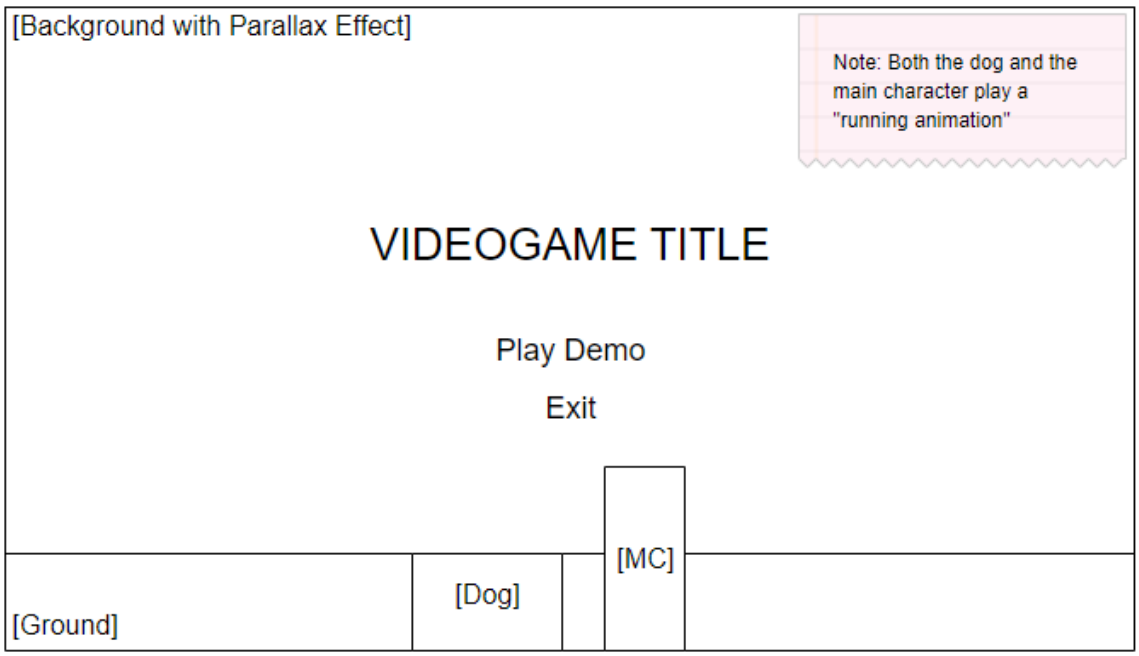


Ilustración 19. Prototipo de interfaz del menú principal. Fuente: Elaboración propia, 2024

MENÚ DE PAUSA: Este menú ofrece tres opciones principales: "Resume" para continuar la partida, "Character Stats" para visualizar las estadísticas del personaje, y "Main Menu" para salir de la partida y volver al menú principal (ver Ilustración 20). En esta fase del desarrollo, el menú no incluye opciones de configuración ni un sistema de inventario, ya que estas funcionalidades están previstas para futuras iteraciones del proyecto.

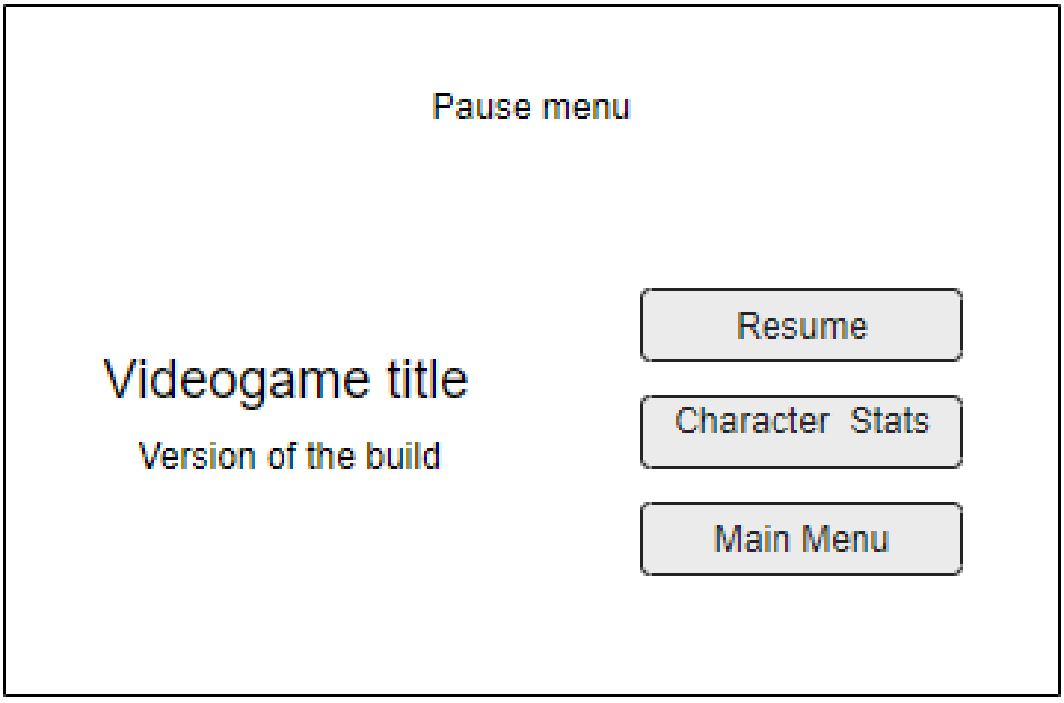


Ilustración 20. Prototipo de interfaz del menú de pausa. Fuente: Elaboración propia, 2024

VENTANA DE ESTADÍSTICAS DEL JUGADOR: En esta ventana de "Character Stats" (ver Ilustración 21) se presentará, en la parte izquierda, una animación o imagen del personaje del jugador, mostrando su aspecto actual o la armadura equipada. A la derecha, se desplegará un listado vertical con las habilidades del personaje y sus respectivas puntuaciones. En esta demo, las habilidades se limitan a dos: salud y fuerza. Sin embargo, se prevé la inclusión de más habilidades en futuras iteraciones del proyecto.

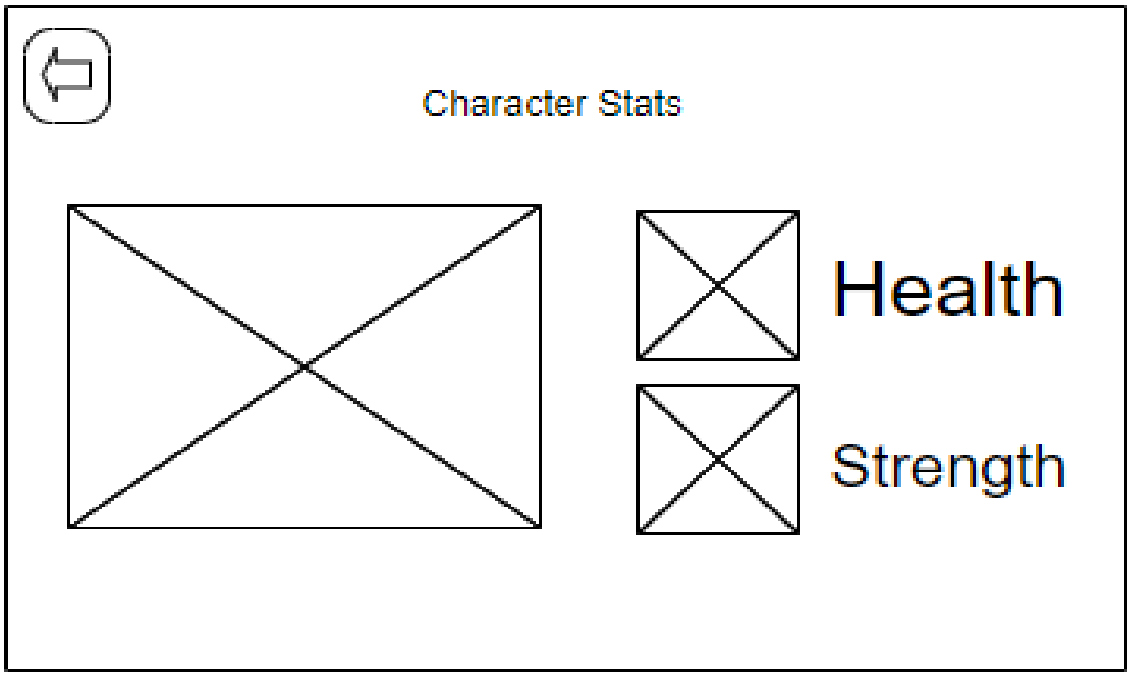


Ilustración 21. Prototipo de interfaz de la ventana de estadísticas del personaje principal. Fuente: Elaboración propia, 2024

6.3. Prototipo de mapa y diseño final de la primera iteración

En la ilustración 22 se muestra un prototipo detallado de la primera zona del videojuego, que es una de las cinco zonas planeadas en el proyecto completo.

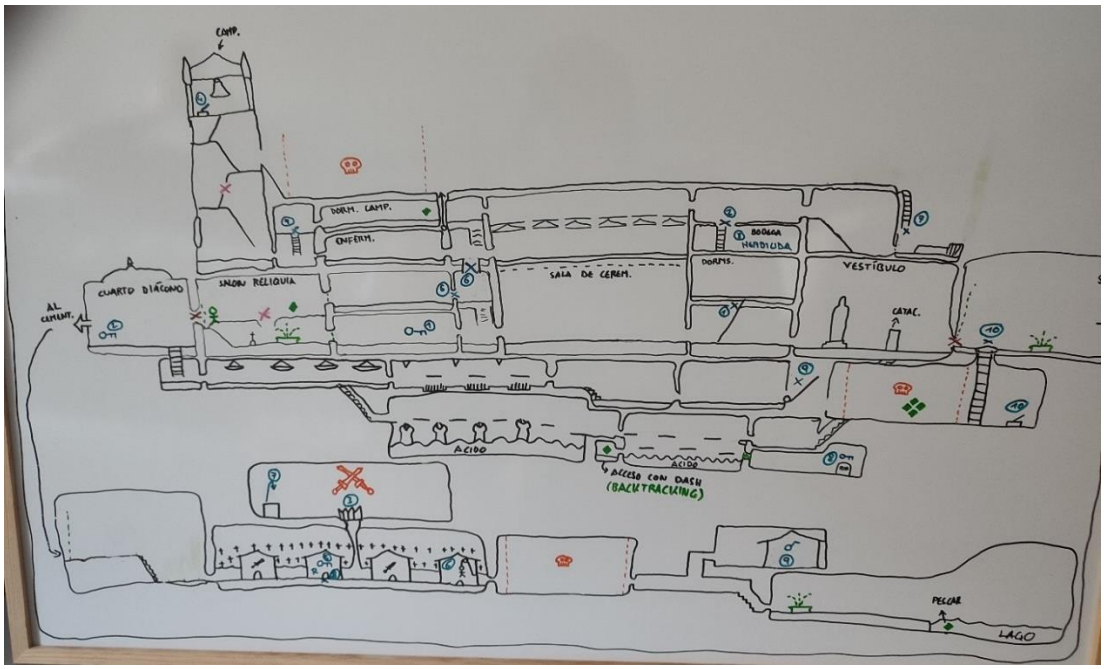


Ilustración 22. Prototipo de mapa para la primera zona del videojuego. Fuente: Elaboración propia, 2024

Sin embargo, debido a las limitaciones de tiempo y la complejidad inherente al desarrollo, se decidió reducir el alcance del proyecto y diseñar una zona más sencilla, representada en la ilustración 23, la cual servirá como una zona de introducción y tutoriales en la versión final del producto.

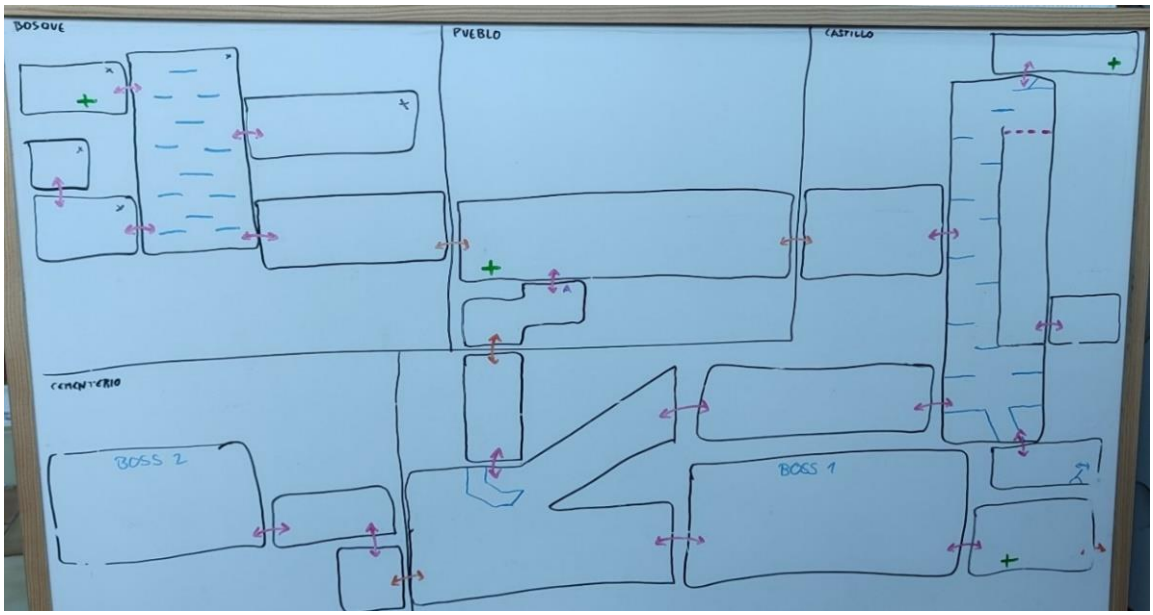


Ilustración 23. Prototipo de mapa para la zona introductoria del videojuego. Fuente: Elaboración propia, 2024

Esta nueva área no solo funcionará como una zona introductoria previa a la primera, sino que también servirá para la demo técnica, permitiendo comprobar y ajustar las funcionalidades más esenciales del videojuego antes de continuar con el desarrollo del resto de las zonas. Tras hacer unos ajustes al prototipo anterior y eliminar una de las partes (ya que era de un tamaño tan reducido que no tenía sentido incorporarlo), se han recopilado

un conjunto de recursos de internet y se han diseñado las diferentes áreas, las cuales pueden observarse en el siguiente capítulo.

CAPÍTULO 7:
IMPLEMENTACIÓN DEL CÓDIGO

En este capítulo, se abordará la implementación del código del proyecto, comenzando con una explicación general de los nodos más utilizados y su configuración de manera genérica. Esta sección proporcionará una comprensión básica de cómo se estructuran y configuran los elementos clave del proyecto en Godot. A continuación, se procederá con un análisis más detallado de las escenas principales, explicando los nodos que las componen y el propósito de cada uno en el contexto del juego. Para cada escena, se describirán los métodos incluidos en el script asociado, detallando su funcionamiento general y su contribución al flujo de juego. Cabe destacar que no se incluirá el código fuente en esta sección (a excepción del fichero General) para facilitar la lectura y comprensión del contenido; sin embargo, aquellos interesados en explorar el proyecto con mayor profundidad podrán acceder al repositorio completo en GitHub mediante el siguiente enlace:

- https://github.com/mgarper/Codename-Dream

7.1. Explicación de la estructura del proyecto

En primer lugar, se explica la estructura del proyecto (ver Ilustración 24), la cual está organizada de manera que facilita la gestión y el desarrollo de las diferentes partes del videojuego. Las diferentes carpetas que conforman el proyecto son:

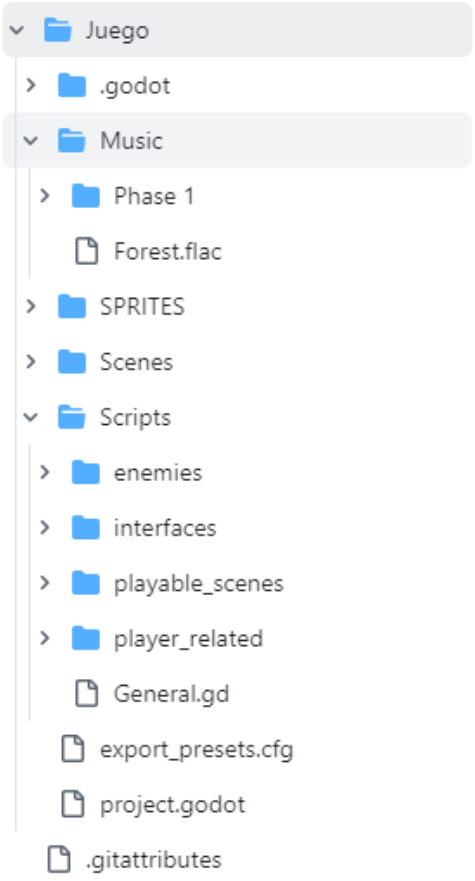


Ilustración 24. Estructura del proyecto en GitHub. Fuente: Elaboración propia, 2024

1. **.godot**: Esta carpeta generalmente contiene archivos de configuración y metadatos específicos del motor Godot. Se utiliza para almacenar configuraciones internas que ayudan a gestionar el proyecto dentro del entorno de desarrollo.
2. **Music**: Esta carpeta almacena los archivos de música utilizados en el videojuego. Dentro de esta carpeta, hay una subcarpeta llamada "Phase 1", que contiene las pistas de audio específicas para la primera fase del juego.
3. **Sprites**: Esta carpeta contiene los archivos de imágenes que se utilizan para los gráficos y las animaciones del juego, como personajes, enemigos, objetos y fondos.
4. **Scenes**: Esta carpeta contiene las diferentes escenas del juego, como los niveles, menús y otras partes interactivas del proyecto. Cada archivo dentro de esta carpeta representa una escena con la que el jugador puede interactuar.
5. **Scripts**: Dentro de esta carpeta, se encuentran varios subdirectorios que organizan los scripts del juego por categorías:
    - *enemies*: Scripts relacionados con el comportamiento y las características de los enemigos del juego.
    - *interfaces*: Scripts que manejan las interfaces de usuario, como menús, HUDs (Heads-Up Display), y otras interacciones de la interfaz gráfica.
    - *playable_scenes*: Scripts específicos para las escenas jugables, controlando aspectos como la lógica del juego y la interacción del jugador dentro de esas escenas (zonas jugables de la demo).
    - *player_related*: Scripts que gestionan las acciones y atributos del jugador, como el movimiento, la herramienta de guardado y las estadísticas.
6. **General.gd**: Es un script general que probablemente contiene funciones o métodos globales que se utilizan en todo el proyecto, como la gestión del estado del juego, el manejo de eventos comunes, y otras utilidades que deben ser accesibles desde diferentes partes del juego.

7.2. Explicación general y configuración de nodos comunes en Godot

A continuación, se va a describir cuales son los nodos y conjuntos de nodos más comunes del proyecto, así como una breve explicación de cómo pueden configurarse:

- **Sprite2D**: Es un nodo que permite establecer imágenes en pantalla. Si la imagen solo contiene un fotograma, no es necesario realizar una configuración previa. En caso contrario, como en la ilustración 25, se deben configurar los atributos *H_frames* y *V_frames* (ver Ilustración 26), que determina el número de fotogramas verticales y horizontales que hay en la imagen, de manera que tras realizar la modificación únicamente se verá uno de los fotogramas de la imagen.

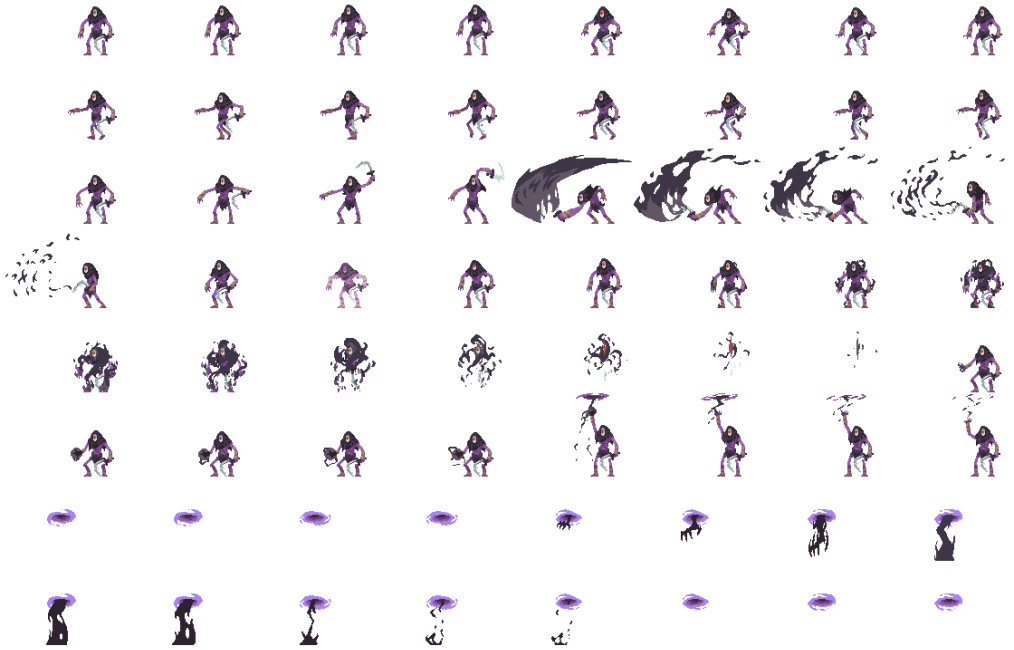


Ilustración 25. Ejemplo de PNG con varios fotogramas en la misma imagen. Fuente: Elaboración propia, 2024

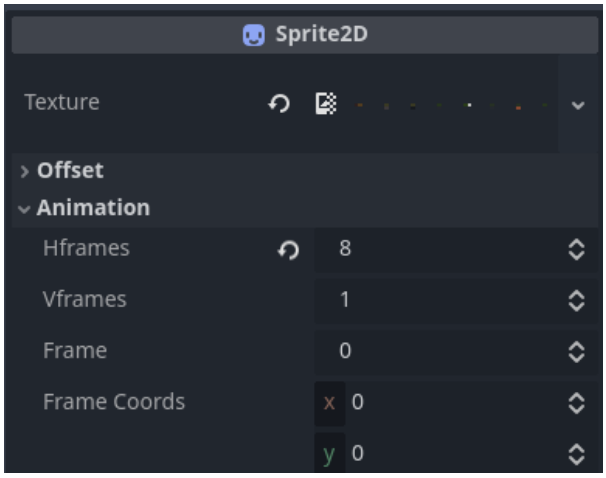


Ilustración 26. Ventana de configuración de un nodo Sprite2D. Fuente: Elaboración propia, 2024

- ***AnimatedSprite/AnimationPlayer***: Ambos son nodos que permiten crear animaciones a partir de un conjunto de fotogramas. La diferencia principal entre ambos es que *AnimatedSprite* necesita que los fotogramas se encuentren en imágenes diferentes, mientras que el *AnimationPlayer* no. Para la configuración del primero solo se tiene que crear la animación y agregar todas las imágenes necesarias, además de configurar la velocidad de ejecución de la animación. Para el segundo es ligeramente más complejo, ya que se sebe emplear la ventana de Animación del editor de escenas de Godot, ubicado en la parte central inferior del editor. Aquí, se crearán diferentes pistas, cada una asociada al cambio de una propiedad dentro de la escena. Por ejemplo, en el caso de la ilustración 27, podemos ver que se modifican 3 propiedades: en primer lugar la propiedad *frame* de *Sprite2D*, que permite determinar que fotograma

de la imagen utilizar en cada instante de tiempo; en segundo lugar la propiedad *h_frames* de *Sprite2D*, con la que establecemos si el nodo se invierte horizontalmente o no; y por último la propiedad *disabled* de *CollisionShape2D*, que nos permite establecer si en un momento determinado el objeto de colisión asociado debe tener la colisión activada o no.

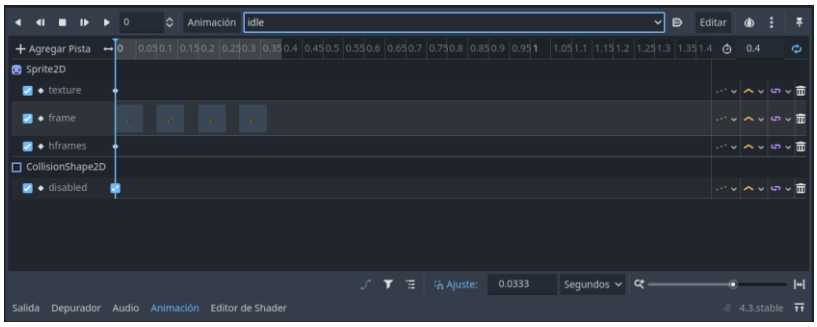


Ilustración 27. Ventana de creación y edición de animaciones para un nodo AnimationPlayer. Fuente: Elaboración propia, 2024

- ***CollisionShape2D***: Es un nodo que permite establecer un objeto de colisión dentro de la escena (ver ilustración 28). Este tipo de nodos no puede funcionar por sí solo, siendo obligatorio que esté anidado dentro de un objeto de tipo *CharacterBody2D, Area2D, StaticBody2D o RigidBody2D*. Para poder emplearlo, es necesario asignarle una forma de colisión, determinando si el nodo tendrá forma circular, cuadrada, ovalada o poligonal. También es posible, en caso de ser necesario, activas la propiedad *One Way Collision*, que permite que el objeto colisione y choque contra otros elementos de la escena únicamente por uno de los lados.

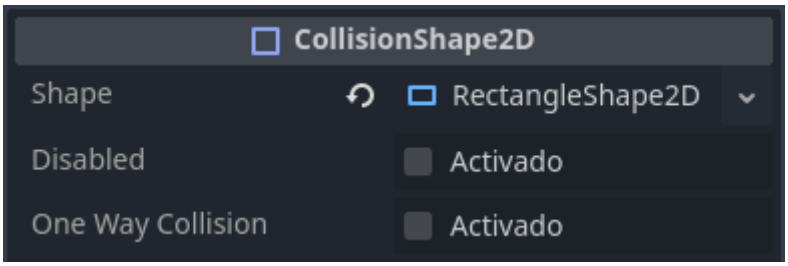


Ilustración 28. Ventana de configuración de un nodo CollisionShape2D. Fuente: Elaboración propia, 2024

- ***Area2D***: Es un nodo que, en el contexto de este proyecto, permite principalmente establecer áreas de detección de objetos de colisión. Para su correcto funcionamiento, es obligatorio establecer un subnodo de tipo *CollisionShape2D*.
- ***RayCast2D***: Es un nodo en forma de flecha que, al igual que el nodo *Area2D*, permite detectar colisiones en la dirección establecida en el editor de escenas. En el contexto de este proyecto se ha empleado para establecer los comportamientos de movimiento de los diferentes enemigos. No necesita ningún tipo de configuración previa.
- ***CanvasLayer***: es un nodo que permite fijar en la pantalla los elementos que se aniden a este nodo, de manera que, por ejemplo, si el personaje se desplaza en pantalla, la posición de los elementos anidados permanece siempre estática y visible en pantalla. En este proyecto no ha necesitado ningún tipo de configuración previa.

- **Conjunto Parallax (*CanvasLayer-ParallaxBackground-ParallaxLayer-Sprite2D*)**: es un conjunto de nodos empleado para crear lo que se conoce como efecto *parallax*, un efecto que da sensación de fondo infinito y con profundidad (ver Ilustración 29). El encargado de gestionar este efecto es el nodo *ParallaxBackground*. Las capas de profundidad se representan mediante nodos *ParallaxLayer*, cada uno conteniendo un Sprite2D que muestra la imagen de fondo correspondiente. Las capas más cercanas al jugador se encuentran más abajo en la jerarquía y se mueven más rápido, creando la ilusión de que están más próximas, mientras que las capas más lejanas se mueven más lentamente. El resultado es un fondo dinámico y atractivo que enriquece la experiencia visual del juego. Para crear este efecto, es necesario realizar tres configuraciones (ver Ilustración 30):

    - Escala en la sección *Parallax Layer*: permite establecer la velocidad de desplazamiento de la capa.
    - Escala en la sección *Node2D*: permite establecer el tamaño de la capa dentro de la escena.
    - *Mirroring*: permite determinar cada cuantos píxeles el *Parallax Background* debe replicar una capa. Para poder calcularlo es necesario tener el ancho en píxeles de la imagen empleada en el *Sprite2D* de la capa y la escala de la sección *Node2D*. El valor del *mirroring* se obtiene multiplicando los dos valores previos.

Ilustración 29. Estructura de un Conjunto Parallax formado por 3 capas de fondos. Fuente: Elaboración propia, 2024

Ilustración 30. Ventana de configuración de un nodo Parallax Layer. Fuente: Elaboración propia, 2024

- ***Camera2D***: es un nodo que permite establecer una cámara personalizada dentro de la escena donde se ubique, posicionando el *viewport* de la escena en el lugar donde se encuentre la cámara. No necesita una configuración previa, pero existen cuatro características (ver Ilustración 31) que en ocasiones puede ser conveniente modificar:

    - Escala: es una propiedad que permite establecer el nivel de *zoom* de la cámara dentro de la escena.
    - *Position Smoothing*: es una propiedad que permite establecer si el movimiento de la cámara es más o menos suave.
    - Ubicación dentro del árbol de nodos de la escena: no es una propiedad como tal, pero la posición dentro del árbol de nodos puede influir en cómo se visualiza el juego al ejecutarlo. Por ejemplo, si se posiciona en la escena, sin anidar a otro nodo, la cámara mostrará de manera fija la zona que hayamos establecido en el editor de escenas; por otro lado, si se anida sobre un nodo en movimiento (por ejemplo, un enemigo) la cámara seguirá el movimiento del propio nodo padre.
    - Límites: permite determinar los límites de movimiento de la cámara, de manera que se en el momento que la cámara alcanza un determinado píxel de posición, se queda fija.

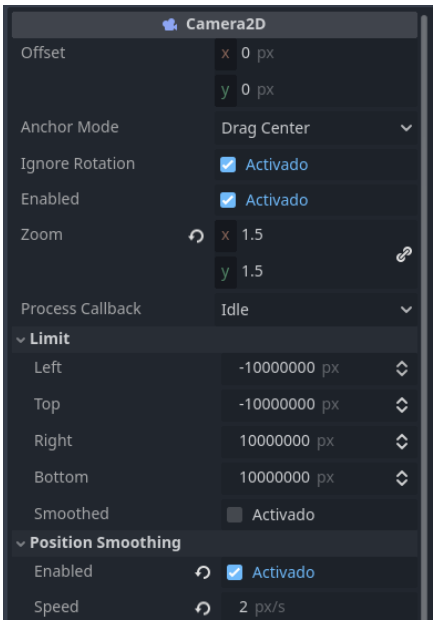


Ilustración 31. Ventana de configuración de un nodo Camera2D. Fuente: Elaboración propia, 2024

- ***VBoxContainer***: es un nodo que permite anidar de manera vertical todos los nodos que se establezcan como hijos. En este proyecto no ha necesitado ningún tipo de configuración previa.
- ***Label***: es un nodo que permite colocar etiquetas de texto dentro de la escena. No necesita ningún tipo de configuración más allá del contenido de texto que debe mostrar la etiqueta, aunque si se desea emplear una fuente externa de un estilo concreto, puedo modificarse insertando la fuente importada en la sección *Control/ThemeOverrides/Fonts* (ver Ilustración 32).

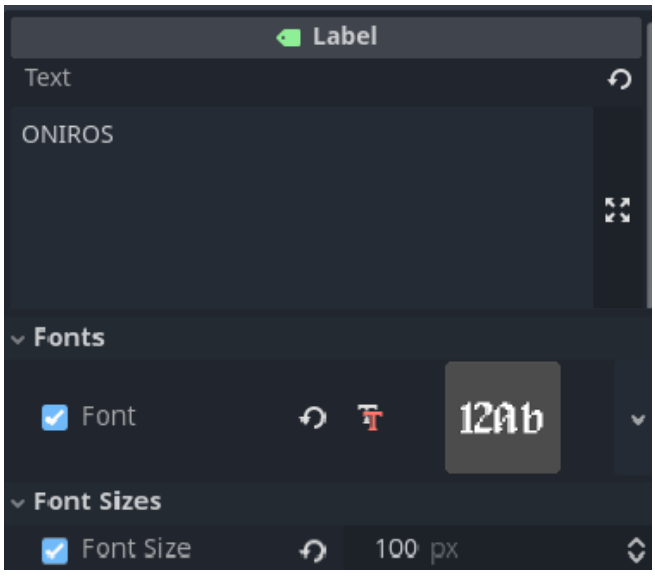


Ilustración 32. Ventana de configuración de un nodo Label. Fuente: Elaboración propia, 2024

- ***Button***: es un nodo que permite colocar botones en una escena. No necesita ningún tipo de configuración previa, aunque si se pueden agregar textos, iconos o efectos ante

una serie de acciones (por ejemplo, cambiar de color la fuente del texto del botón al pasar el ratón por encima, como se ve en la ilustración 33).

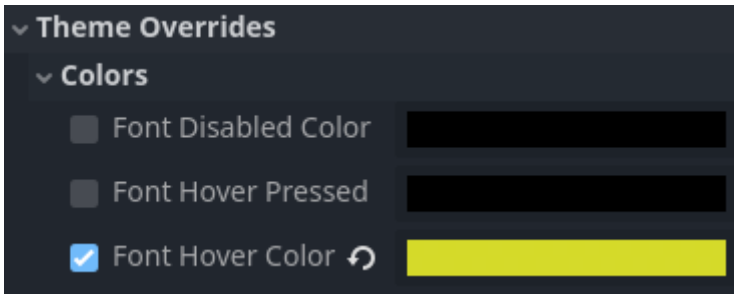


Ilustración 33. Ventana de configuración de un nodo Button. Fuente: Elaboración propia, 2024

- ***Conjunto efecto de transición (CanvasLayer – ColorRect - AnimationPlayer)***: se trata de un conjunto de nodos utilizado para generar las animaciones de fundido a negro (y viceversa) durante las transiciones entre escenas (ver Ilustración 34). Este conjunto incluye el nodo CanvasLayer, que asegura que los nodos hijos se mantengan visibles en la pantalla principal sin ser afectados por el movimiento del jugador; el nodo ColorRect, que establece un fondo de color (en este caso, negro); y un nodo AnimationPlayer, que permite crear las animaciones de aparición y desaparición del fondo oscuro. La configuración se realiza mediante dos acciones: primero, se ajusta el valor del atributo "a" en la propiedad "modulate" a 0, controlando así la transparencia del color (ver Ilustración 35). Luego, se crean dos animaciones: una para la aparición del fondo (donde el atributo "a" pasa de 0 a 255) al salir de una escena (ver Ilustración 36), y otra para la desaparición (donde el atributo pasa de 255 a 0) al cargar la nueva escena.

Ilustración 34. Estructura de un conjunto de efecto de transición. Fuente: Elaboración propia, 2024

Ilustración 35. Ventana de configuración de un nodo ColorRect. Fuente: Elaboración propia, 2024

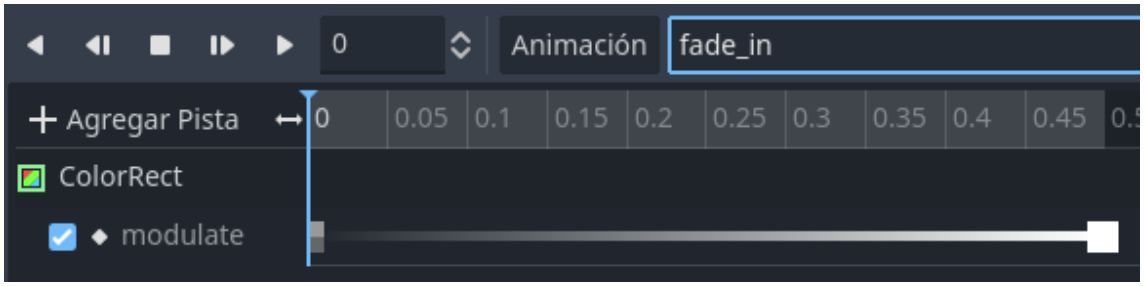


Ilustración 36. Ejemplo de animación de salida de escena del conjunto de efecto de transición. Fuente: Elaboración propia, 2024

- ***AudioStreamPLayer***: es el nodo que permite insertar pistas de audio y música dentro de una escena. En este caso se han utilizado para poner música de fondo en las escenas a modo de banda sonora, por lo que para configurarlo únicamente se ha tenido que insertar la pieza musical que se deseaba reproducir (ver Ilustración 37) y establecer el atributo de *loop* a *true*, para que en el momento que se llegue al final de la pista de audio ésta vuelva a reporducirse.

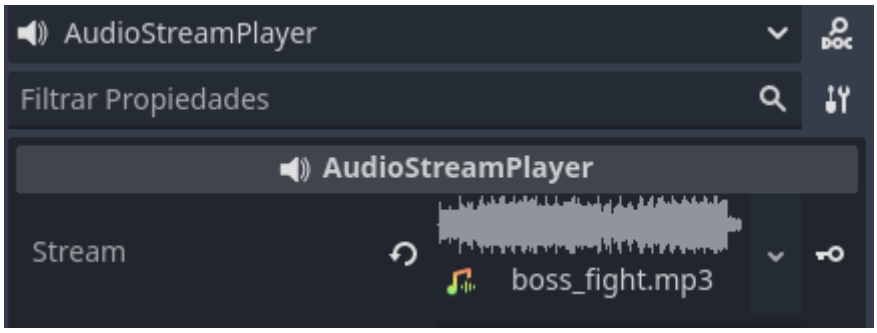


Ilustración 37. Ventana de configuración de un nodo AudioStreamPlayer. Fuente: Elaboración propia, 2024

7.3. Fichero global *autoload*

El archivo *General.gd* es un script global que se utiliza para conservar datos entre los cambios de escena y manejar la persistencia de estos durante la sesión del juego, lo que en Godot se conoce como un fichero global *autoload*. Sus principales funcionalidades son las siguientes:

- ***change_scene***: es responsable de cambiar la escena actual a otra específica, gestionando parámetros como si es la primera carga, si se está cargando un juego guardado, o si el jugador ha muerto. Este método también recibe la escena de origen y destino, así como las direcciones para establecer la posición del jugador en la nueva escena. Dependiendo de estos parámetros, se carga la escena correspondiente, y en el caso de que la escena origen sea "demo_end", se elimina el archivo de guardado existente, de manera que, si juegan de nuevo a la demo, se inicie una nueva partida desde cero.
- ***_input***: se encarga de detectar las entradas del jugador. Específicamente, si el jugador presiona la tecla de pausa, el método llama a *_open_menu* para abrir el menú de pausa.
- ***_open_menu***: carga e instancia el menú de pausa del juego. Este método carga la escena correspondiente al menú de pausa (*pause_menu.tscn*), la instancia y la añade como un hijo al nodo del jugador. Además, posiciona el menú en la pantalla y luego limpia la referencia a la variable *pause_menu* después de su uso.
- ***_check_character_stats*:** muestra la ventana de estadísticas del personaje. De manera similar al método *_open_menu*, carga la escena de estadísticas del personaje (*character_stats.tscn*), la instancia, la añade al nodo del jugador y la posiciona en la pantalla para permitir al jugador revisar las estadísticas.
- ***set_player_attributes***: se encarga de establecer los atributos del jugador, tales como el último lugar visitado, la vida máxima, la fuerza máxima, los puntos actuales, el nivel, el progreso en el castillo y la vida actual. Este método se utiliza tanto para inicializar como para actualizar los atributos del jugador que deben persistir durante la sesión de juego.
- ***update_points***: es utilizado para actualizar los puntos actuales del jugador, sumando la cantidad de puntos especificados a los puntos que el jugador ya ha acumulado.

- ***retry***: permite reiniciar el juego en caso de que el jugador quiera intentarlo de nuevo tras morir. Este método desactiva las banderas first_load y load_game, libera todos los

nodos que pertenecen al grupo Persist, carga el archivo de guardado (*savegame.save*), y restaura la escena y los atributos del jugador según la información guardada en ese archivo.

7.4. Menús e interfaces

A continuación, se comienzan a describir las escenas implementadas en esta demo técnica, comenzando en primer lugar por las escenas asociada a interfaces y menús.

- MENÚ PRINCIPAL

Esta escena está asociada al menú principal de la demo (ver Ilustración 38), la cual se carga en el momento que se ejecuta el videojuego. Los nodos que componen esta ventana (ver Ilustración 39) son los siguientes:

Ilustración 38. Diseño para la demo del menú principal. Fuente: Elaboración propia, 2024

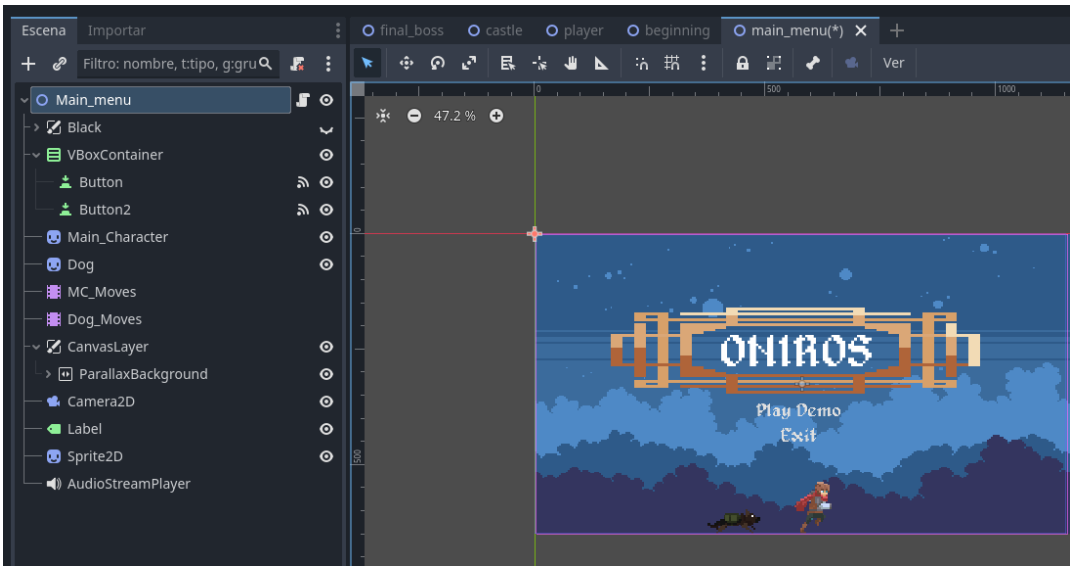


Ilustración 39. Escena del menú principal. Fuente: Elaboración propia, 2024

- *Black*: es un conjunto de nodos del tipo efecto de transición.
- *VBoxContainer*: este nodo cuenta con dos subnodos, que corresponden a los botones de "Play Demo" y "Exit", uno para empezar la partida (sea nueva o cargada) y otro para detener el juego, respectivamente.
- *Main_Character*: es un nodo de tipo *Sprite2D*. En este caso el diseño corresponde al del personaje protagonista.
- *Dog*: nodo de igual tipo que Main_Character. En este caso el diseño corresponde al del perro que acompaña al personaje protagonista (que no está presente en esta demo).
- *MC_Moves*: es un nodo de tipo *AnimationPlayer* que se encarga de las animaciones del personaje principal.
- *Dog_Moves*: es un nodo de tipo *AnimationPlayer* que se encarga de las animaciones del perro.
- *CanvasLayer*: es el nodo que agrupa los componentes necesarios para realizar el efecto *parallax* descrito anteriormente.
- *Camera2D*: es el nodo que crea una cámara fija que muestre únicamente lo que alcanza a ver el recuadro morado de la imagen.
- *Label*: es el nodo empleado para poner el título del videojuego en pantalla.
- *Sprite2D*: es el nodo correspondiente al marco dorado que se encuentra alrededor del título.
- *AudioStreamPlayer*: es un nodo empleado para reproducir una pista de música en bucle

Analizando el script asociado a esta escena (*main_menu.gd*) observamos las siguientes funcionalidades:

- ***_ready***: configura la ventana en modo ventana y se inician las animaciones de los personajes en el fondo. Además, se preparan las capas de parallax para crear un efecto de desplazamiento en el fondo y se establece el foco inicial en el primer botón del menú.
- ***_process***: se ejecuta en cada fotograma y se encarga de mover las capas de parallax, lo que da una sensación de profundidad dinámica en el fondo del menú principal.
- ***_input***: detecta cuando el usuario presiona la acción de "accept" (establecida en el mapa de entrada del proyecto). Según el botón que tenga el foco en ese momento, emite la señal de *pressed* para activar la acción correspondiente, permitiendo al jugador interactuar con el menú.
- ***_on_button_pressed***: método asociado a la señal *pressed* emitida por el botón de "Play Demo". Llama al método *load_game*.
- ***_on_button_2_pressed***: método asociado a la señal *pressed* emitida por el botón de "Exit". Cierra el juego, parando así la ejecución del programa.
- ***load_game***: es responsable de gestionar la carga del juego. Si no existe un archivo de guardado, el juego comienza desde el inicio con atributos predeterminados. Sin embargo, si hay un archivo de guardado, este método carga los datos del jugador y cambia la escena al último punto guardado.

- MENÚ DE PAUSA

Esta escena está asociada al menú de pausa de la demo (ver Ilustración 40), la cual se carga en el momento que el jugador pulsa el botón de pausa en cualquiera de los diferentes dispositivos de entrada. Su nodo raíz es de tipo *CanvasLayer* para asegurar que el menú se muestre centrado en pantalla y no se vea afectado por el movimiento del jugador. Los nodos que componen esta ventana (ver Ilustración 41) son los siguientes:

Ilustración 40. Diseño para la demo del menú de pausa. Fuente: Elaboración propia, 2024

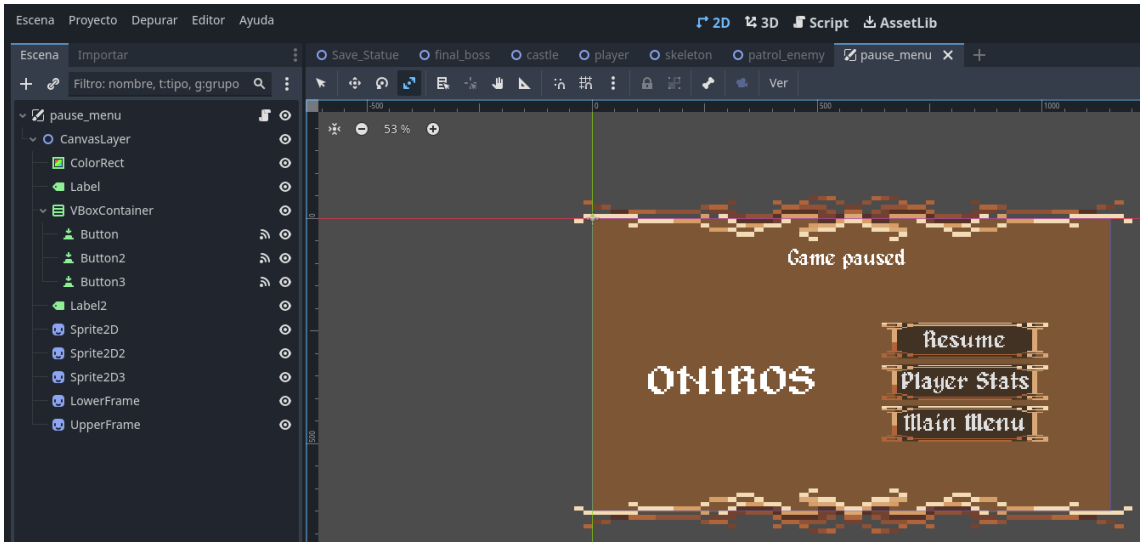


Ilustración 41. Escena del menú de pausa. Fuente: Elaboración propia, 2024

- *ColorRect*: este nodo permite poner el color de fondo del menú.
- *Label*: etiqueta correspondiente al texto “Oniros”.
- *VBoxContainer*: es el nodo que contiene alineados verticalmente los tres botones de interacción del menú de pausa: uno para continuar la partida, otro para abrir la ventana de consulta de estadísticas y el último para volver al menú principal.
- *Label2*: etiqueta correspondiente al título de la ventana (“Game paused”).
- *Sprite2D, Sprite2D2 y Sprite2D3*: nodos correspondientes a los marcos que rodean los botones.
- *LowerFrame y UpperFrame*: nodos de tipo *Sprite2D* utilizados para mostrar los diseños de los bordes superiores e inferiores de la ventana.

Analizando el script asociado a esta escena (*pause_menu.gd*) observamos las siguientes funcionalidades:

- ***_ready***: se ejecuta cuando el nodo al que está adjunto el script se ha añadido al árbol de nodos. En este caso, el método se encarga de enfocar automáticamente el primer botón del menú de pausa, asegurando que el jugador pueda interactuar con él usando el teclado o el controlador. Además, verifica si el juego está en pausa; si no lo está, lo pausa al establecer la propiedad *paused* del árbol de nodos en true.
- ***_input***: gestiona la entrada del jugador. Específicamente, detecta si el jugador ha presionado la tecla o botón de "aceptar" (usualmente el botón de acción principal en un juego). Dependiendo de cuál de los tres botones del menú de pausa tiene el enfoque en ese momento (Resume, Player Stats, o Main Menu), se emite la señal pressed para ejecutar la acción correspondiente.
- ***_on_button_pressed***: se activa cuando el jugador selecciona la opción de reanudar el juego (el botón "Resume"). Cuando se llama, este método invierte el estado de pausa del juego (reanuda la partida) y cierra el menú de pausa eliminando el nodo asociado a este menú.
- ***_on_button_2_pressed***: se activa cuando el jugador selecciona la opción de ver las estadísticas del personaje (el botón "Player Stats"). Este método llama a la función *General.check_character_stats*, que muestra la ventana de estadísticas del personaje, y luego cierra el menú de pausa.
- ***_on_button_3_pressed***: se activa cuando el jugador selecciona la opción de regresar al menú principal (el botón "Main Menu"). Este método despausa el juego, obtiene la ruta del nodo padre para determinar la escena actual, y luego llama a la función *General.change_scene* para cambiar la escena al menú principal, cerrando el menú de pausa en el proceso.

- ESTADÍSTICAS DEL PERSONAJE

Esta escena está asociada al menú de visualización de estadísticas de la demo (ver Ilustración 42), la cual se carga en el momento que el jugador pulsa el botón “player stats” del menú de pausa. Su nodo raíz es de tipo *CanvasLayer* para asegurar que el menú se muestre centrado en pantalla y no se vea afectado por el movimiento del jugador. Los nodos que componen esta ventana (ver Ilustración 43) son los siguientes:

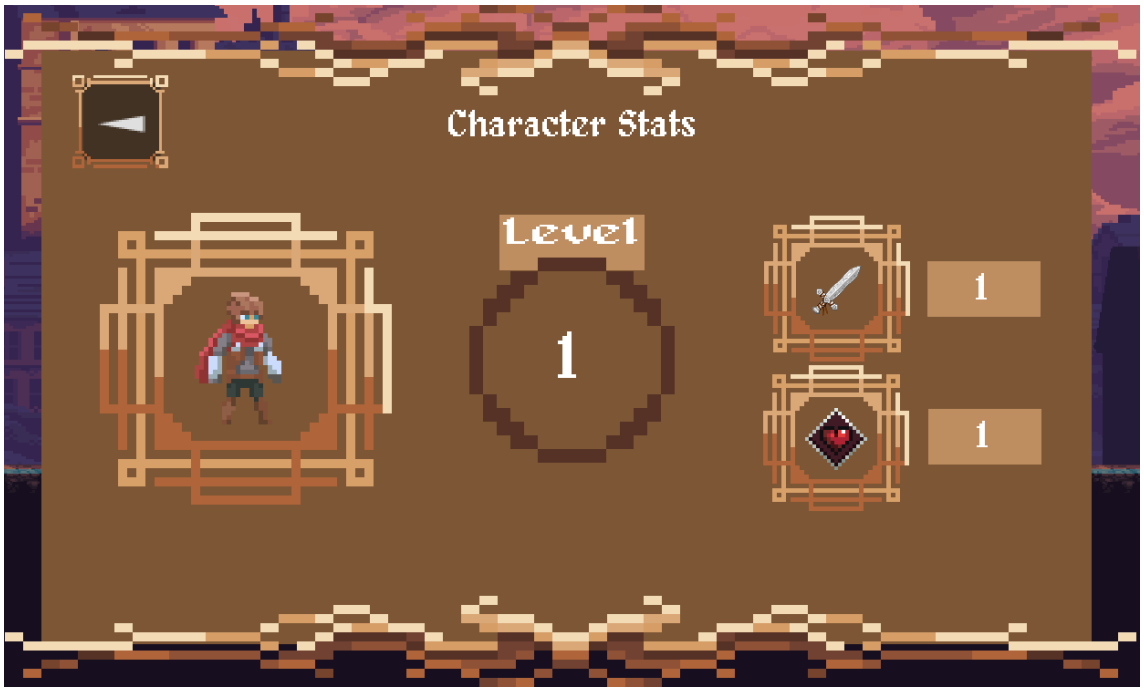


Ilustración 42. Diseño para la demo de la ventana de estadísticas del personaje principal. Fuente: Elaboración propia, 2024

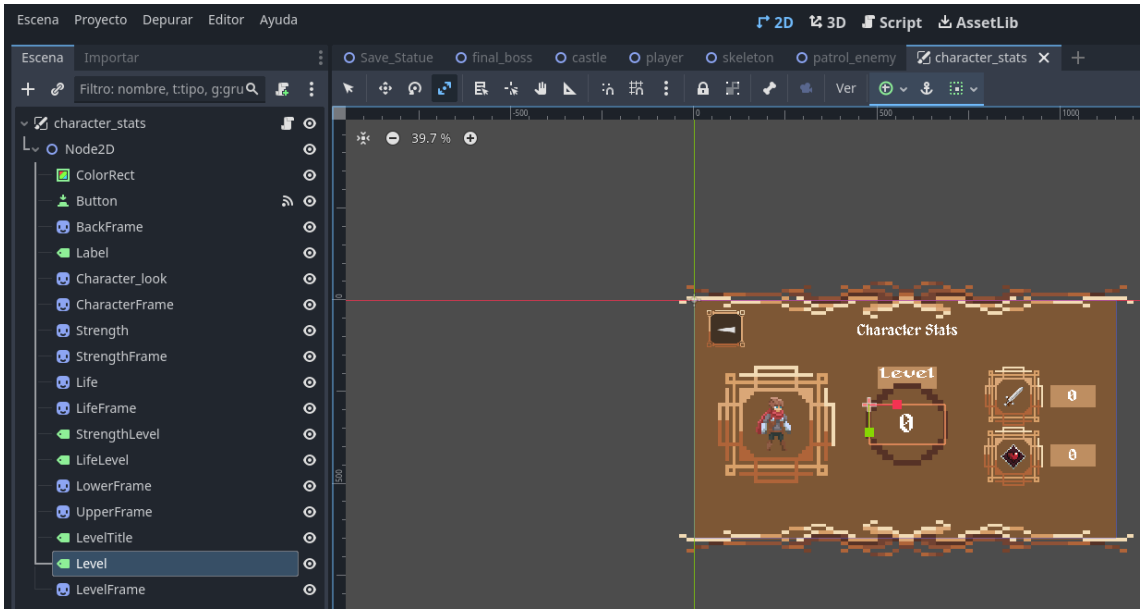


Ilustración 43. Escena del menú de consulta de estadísticas. Fuente: Elaboración propia, 2024

- *ColorRect*: este nodo permite poner el color de fondo del menú.
- *Button*: este nodo corresponde al botón de la esquina superior izquierda, que permite volver al menú de pausa.
- *Backframe*: nodo de tipo *Sprite2D* empleado para mostrar el diseño del marco del botón.
- *Label*: etiqueta correspondiente al título de la ventana (“Character Stats”).

- *Character_look*: nodo de tipo *Sprite2D* que muestra la apariencia actual del personaje.
- *Character_Frame*: nodo de tipo *Sprite2D* empleado para mostrar el diseño del marco del personaje.
- *Strength*: nodo de tipo *Sprite2D* que muestra una representación gráfica del atributo fuerza (espada).
- *StrengthFrame*: nodo de tipo *Sprite2D* empleado para mostrar el diseño del marco del nivel de fuerza.
- *Life*: nodo de tipo *Sprite2D* que muestra una representación gráfica del atributo vida (corazón).
- *LifeFrame*: nodo de tipo *Sprite2D* empleado para mostrar el diseño del marco del nivel de vida.
- *StrengthLevel*: etiqueta correspondiente al nivel numérico de fuerza del jugador.
- *LifeLevel*: etiqueta correspondiente al nivel numérico de vida del jugador.
- *LowerFrame* y *UpperFrame*: nodos de tipo *Sprite2D* utilizados para mostrar los diseños de los bordes superiores e inferiores de la ventana.
- *LevelTitle*: etiqueta correspondiente al texto “Level”.
- *Level*: etiqueta correspondiente al nivel numérico del jugador.
- *LevelFrame*: nodo de tipo *Sprite2D* empleado para mostrar el diseño del marco del nivel del nivel del jugador.

Analizando el script asociado a esta escena (*character_stats.gd*) observamos las siguientes funcionalidades:

- ***_ready***: se ejecuta cuando el nodo al que está adjunto el script se ha añadido al árbol de nodos. En este caso, el método se asegura de que el botón de retorno ("Back") obtenga automáticamente el foco para que el jugador pueda interactuar con él. Además, el método actualiza los textos de las etiquetas que muestran el nivel del personaje, su fuerza y su vida máximas, utilizando los valores almacenados en el script global General.
- ***_input***: se encarga de gestionar la entrada del jugador. Específicamente, detecta si el jugador ha presionado la tecla o botón de "aceptar" (usualmente el botón de acción principal en un juego). Si el botón de retorno tiene el foco, se emite la señal pressed para activar la función asociada.
- ***_on_button_pressed***: se activa cuando el jugador selecciona el botón de retorno. Este método llama a la función *General.open_menu*, que reabre el menú de pausa. Luego, cierra la ventana de estadísticas del personaje eliminando el nodo asociado a esta ventana.

- FIN DE LA VERSIÓN DE DEMOSTRACIÓN

Esta escena está asociada al menú de finalización de la demo, la cual se carga en el momento que el jugador cruza el portal que hay abierto en la escena del castillo (se explicará más adelante). Anidado a su nodo raíz hay un nodo de tipo *CanvasLayer* para asegurar que el menú ocupe toda la pantalla. Los nodos que componen esta ventana (ver Ilustración 44) son los siguientes:

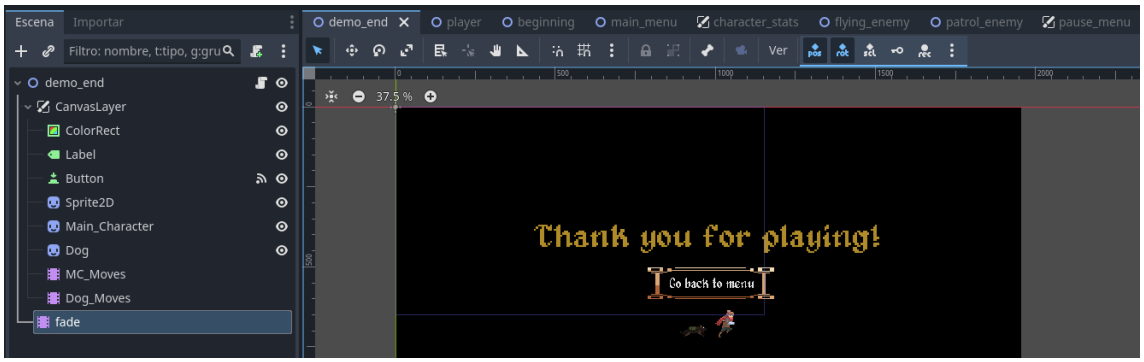


Ilustración 44. Escena del menú de fin de la demo. Fuente: Elaboración propia, 2024

- *CanvasLayer*, *ColorRect* y *fade*: conjunto de nodos que permiten realizar la aparición del fondo negro, así como del resto de elementos en pantalla de manera gradual, como si de una transición se tratara.
- *Label*: etiqueta correspondiente al texto “Thank you for playing!”.
- *Button*: botón para volver al menú principal.
- *Sprite2D*: marco del botón de retorno al menú principal.
- *Main_Character* y *Dog*: nodos de tipo *Sprite2D* para mostrar los diseños del personaje principal y su perro.
- *MC_Moves* y *Dog_Moves*: nodos de tipo *AnimationPlayer* para realizar las animaciones de movimiento del protagonista y su perro.

Analizando el script asociado a esta escena (*demo_end.gd*) observamos las siguientes funcionalidades:

- ***_ready***: se ejecuta cuando el nodo entra en el árbol de la escena por primera vez. En este método, la ventana del juego se configura en modo pantalla completa con *DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_FULLSCREEN)*. Luego, se obtienen y asignan las referencias a varios nodos de la escena, como el botón, la etiqueta, el sprite del botón, el personaje principal (*Main_Character*), el perro (*Dog*), las animaciones de movimiento de ambos personajes, y el nodo de la animación de fundido (*fade*). Posteriormente, se deshabilita el botón, y se establece la transparencia total (*modulate.a = 0*) para la etiqueta, el botón, el *sprite* del botón, el personaje principal, y el perro. Las animaciones de correr para el personaje principal y el perro se inician, y se reproduce la animación de fundido de entrada. Una vez que la animación de fundido ha finalizado, el botón se habilita y adquiere el foco.
- ***_input***: se encarga de detectar cuando el jugador presiona la tecla o botón de "aceptar" del mapa de entrada. Si el botón tiene el foco en ese momento, se emite la señal pressed, que activa la función correspondiente.
- ***_on_button_pressed***: se ejecuta cuando el jugador selecciona el botón para volver al menú principal. Este método llama a la función *General.change_scene* del script global *General.gd*, lo que cambia la escena actual (*demo_end*) a la escena del menú principal (*main_menu*).

7.5. Escenas instanciables

En este apartado se recogerán aquellas escenas que pueden emplearse como objetos en otras escenas, de ahí que se denominen escenas instanciables.

7.5.1. Jugador principal: *PLAYER*

Esta escena está asociada al personaje principal, el cual el jugador puede manejar durante toda la partida. Esta escena se instancia cada vez que se inicia una nueva partida, se carga un espacio de guardado o si se cambia de escena. Los nodos que componen esta ventana (ver Ilustración 45) son los siguientes:

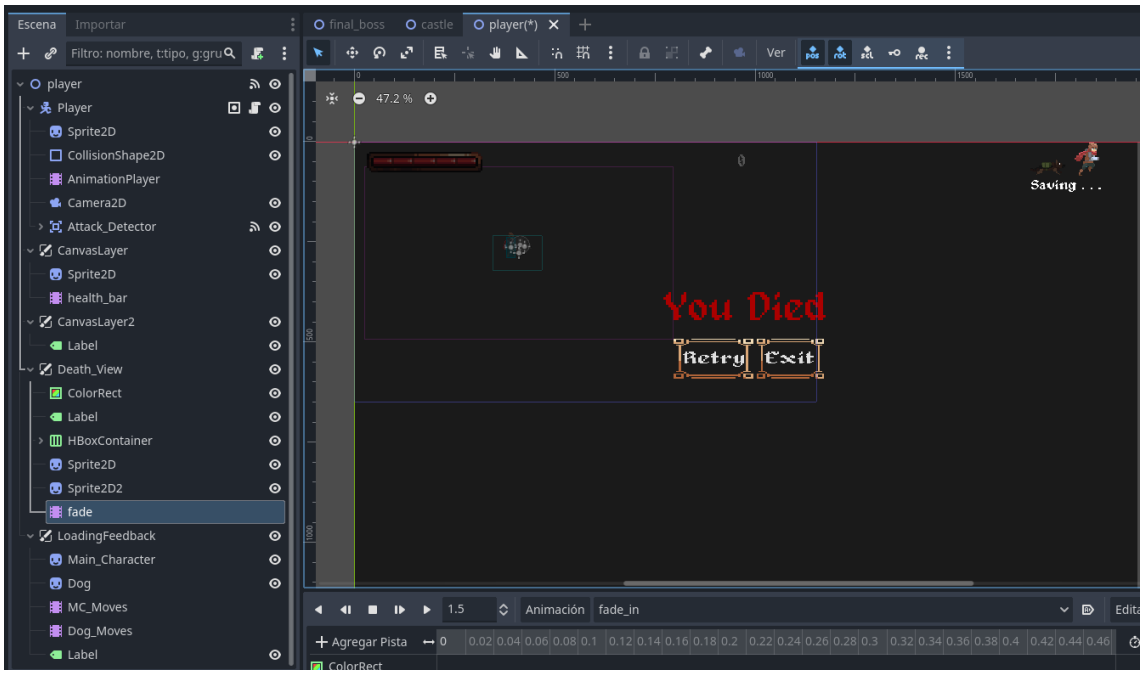


Ilustración 45. Escena del personaje jugable. Fuente: Elaboración propia, 2024

- *Player*: es un nodo de tipo *CharacterBody2D* (ver Ilustración 46). A este nodo, se le han añadido los siguientes nodos hijos: en primer lugar, un nodo *Sprite2D*, para aplicar el diseño del personaje; en segundo lugar, un nodo *CollisionShape2D* rectangular que permite definir la colisión del cuerpo del jugador; en tercer lugar, un nodo *AnimationPlayer* para gestionar las animaciones del jugador; a continuación, un nodo *Camera2D* para añadir una cámara que siga al jugador mientras se desplaza por la escena; y por último un nodo de tipo *Area2D* (*Attack_Detector*) que crea el área de ataque del jugador para dañar a los enemigos.
- *CanvasLayer y CanvasLayer2*: son dos nodos que permiten mostrar de manera fija en pantalla los elementos correspondientes al HUD del jugador, siendo este la barra de vida y la puntuación obtenida tras derrotar enemigos.
- *Death_View*: es un nodo que cuenta con un fondo oscuro, un *label* y dos botones, empleado principalmente para aparecer cuando la vida del jugador llega a cero, permitiéndole decidir si cargar el último punto de guardado o salir de la partida.
- *LoadingFeedback*: nodo que muestra de manera fija en pantalla un *label* con el texto “Saving ...” y la animación del protagonista y su perro corriendo, ofreciendo

una retroalimentación visual de la interacción del jugador con la estatua (muestra que el jugador está guardando partida).

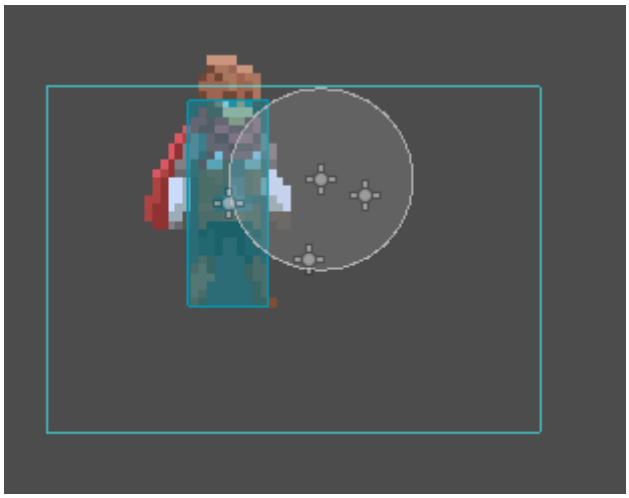


Ilustración 46. Protagonista con sus objetos de colisión en la escena del personaje jugable. Fuente: Elaboración propia, 2024

Analizando el script asociado a esta escena (*player.gd*) observamos las siguientes funcionalidades:

- ***_ready***: se ejecuta cuando el nodo ingresa al árbol de la escena. Inicializa varias variables asociadas con los elementos de la interfaz de usuario (como la barra de salud y el menú de muerte), configura las animaciones del personaje y los elementos asociados, y prepara la interfaz para mostrar la información correcta. El método también configura los valores iniciales de la vida del personaje y establece la visibilidad y el estado de los elementos de la interfaz, como el botón de reintento y el feedback de carga.
- ***_physics_process***: maneja la física del personaje en cada fotograma. Controla el movimiento del jugador a lo largo del eje horizontal en función de la entrada del jugador (teclas de dirección o joystick), aplica gravedad, gestiona los saltos y el doble salto, y maneja el movimiento de *dash* (deslizamiento rápido). También aplica las animaciones correspondientes dependiendo del estado del personaje (correr, saltar, caer y el *dash*).
- ***_apply_animation***: gestiona las animaciones del personaje en función de su estado actual, como si está corriendo, saltando, cayendo o deslizándose. Cambia la animación actual y ajusta la dirección en la que el *sprite* está mirando (izquierda o derecha). El método también asegura que la animación de *dash* se ejecute completamente antes de permitir que el jugador realice otras acciones.
- ***_input***: maneja la entrada del jugador que no está relacionada directamente con la física, como los ataques y la defensa. Pausa el procesamiento de la física cuando el jugador está realizando una animación de ataque o bloqueando, y lo reanuda una vez que la animación ha terminado. También gestiona la dirección en la que está mirando el *sprite* del jugador según la entrada de movimiento.

- ***damage***: se ejecuta cuando el jugador recibe daño. Reduce la vida del jugador y reproduce la animación correspondiente. Si la vida del jugador llega a cero, se llama al método *dead*.
- ***dead***: maneja la muerte del personaje. Pausa el procesamiento de la física, desactiva la colisión del personaje, reproduce la animación de muerte, y luego muestra la pantalla de muerte con opciones para reintentar o volver al menú principal. También guarda el estado del juego en un archivo.
- ***set*_attributes**: configura los atributos del jugador como vida máxima, fuerza, puntos actuales, nivel y si se ha alcanzado el punto de control tras derrotar al jefe final del castillo. Luego actualiza la interfaz del puntaje en función de los puntos actuales del jugador.
- ***add_points***: suma puntos al total actual y actualiza la interfaz del puntaje, además de actualizar el puntaje en el script global *General.gd*.
- ***update_points***: actualiza el texto que muestra los puntos actuales en la interfaz.
- ***save***: crea un diccionario con los atributos importantes del jugador y lo guarda en el archivo de guardado.
- ***get_life***: simplemente devuelve el valor actual de la vida del jugador.
- ***set_life_frame***: determina el marco de la barra de salud que debe mostrarse dependiendo del nivel de vida del jugador.
- ***restore_life***: restaura la vida del jugador a su valor máximo, reproduciendo la animación de recuperación de salud.
- ***loading_feedback***: muestra una animación de carga con el jugador y el perro corriendo durante un breve período de tiempo.
- ***_on_player_ready***: establece los atributos del jugador según los valores guardados en el script global General cuando el jugador está listo.
- ***_on_detector_body_entered***: se activa cuando otro cuerpo entra en el área de ataque del jugador. Si el cuerpo pertenece al grupo "Enemy" o "BossEnemy", el método llama a la función de golpe del enemigo.
- ***_on_button_pressed*** *y* ***_on_button_2_pressed***: manejan la lógica de los botones en la pantalla de muerte. Permiten al jugador reintentar desde el último punto de guardado o regresar al menú principal.

7.5.2. Enemigos

Enemigo volador: *FLYING_ENEMY*

Esta escena está asociada al enemigo volador, cuyo comportamiento se basa en estar estático volando hasta que detecta al jugador, momento en el que se desplaza para atacarle. Esta escena aparece instanciada en varias escenas, y su número de instancias depende exclusivamente del número de enemigos de este tipo que quieran incluirse en la escena. Esta ventana (ver Ilustración 47) está compuesta principalmente por un nodo de tipo CharacterBody2D, y los nodos que aparecen anidados son los siguientes:

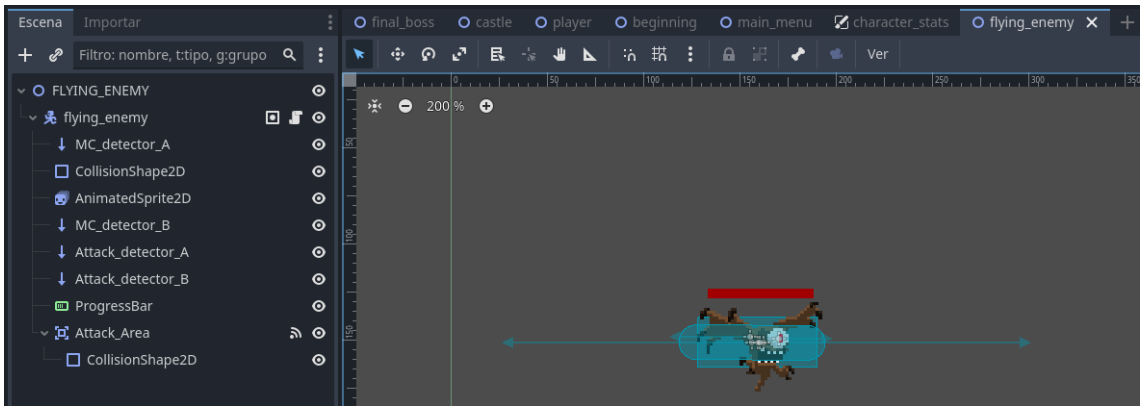


Ilustración 47. Escena del enemigo volador. Fuente: Elaboración propia, 2024

- *MC_Detector A y B, y Attack_detector A y B*: son nodos de tipo RayCast2D que permiten detectar al enemigo para desplazarse y para iniciar la acción de ataque, respectivamente.
- *CollisionShape2D*: nodo empleado para definir la colisión del cuerpo del enemigo.
- *AnimatedSprite2D*: nodo empleado para gestionar las animaciones del enemigo volador.
- *ProgressBar*: nodo que muestra el porcentaje de vida que tiene el enemigo.
- *Attack_Area*: nodo que define el área en que el jugador recibe daño por parte del enemigo volador.

Analizando el script asociado a esta escena (*flying_enemy.gd*) observamos las siguientes funcionalidades:

- ***_ready*:** se ejecuta cuando el nodo entra por primera vez en el árbol de la escena. Aquí se inicializan los nodos que detectan la posición del jugador y los nodos que detectan si el enemigo está en posición de atacar. También se inicializan el *sprite* del enemigo, la barra de salud y el área de colisión de ataque. La barra de salud se oculta inicialmente y se deshabilita el área de colisión para evitar ataques prematuros. Finalmente, se establece la velocidad del enemigo, y las variables que controlan si el enemigo está atacando o moviéndose se configuran en false. El enemigo comienza en su estado de "vuelo".
- ***_physics_process***: se encarga de la lógica de movimiento y ataque del enemigo en cada fotograma. Este método verifica si los detectores de ataque están colisionando con el jugador. Si lo están, el enemigo se detiene y comienza a atacar, activando su área de colisión de ataque. Si no hay colisión, el enemigo sigue moviéndose hacia el jugador si es detectado por los detectores de posición. Si el jugador no está en las cercanías, el enemigo se detiene.
- ***hit***: se ejecuta cuando el enemigo recibe un golpe del jugador. Se reduce el valor de la barra de salud y, si la salud es menor que 100, la barra se vuelve visible. Si la salud del enemigo cae a cero o menos, se llama al método *dead()* para manejar la muerte del enemigo. Además, se reproduce la animación de daño y se espera a que finalice.

- ***dead***: se encarga de la muerte del enemigo. Detiene el procesamiento físico del enemigo, reproduce la animación de muerte, y una vez que esta animación finaliza, se elimina al enemigo de la escena usando *self.queue_free()*.
- ***_on_attack_area_body_entered***: se activa cuando el área de ataque del enemigo entra en contacto con otro cuerpo. Si el cuerpo con el que entra en contacto es el jugador, se ejecuta la función *damage()* en el jugador, infligiéndole daño.

Enemigo terrestre 1: *SKELETON*

Esta escena está asociada al enemigo con apariencia de esqueleto, cuyo comportamiento se basa en patrullar hasta que detecta al jugador, momento en el que se detiene para atacarle. Esta escena aparece instanciada en varias escenas, y su número de instancias depende exclusivamente del número de enemigos de este tipo que quieran incluirse en la escena. Los nodos que componen esta ventana son los que se pueden observar en la ilustración 48.

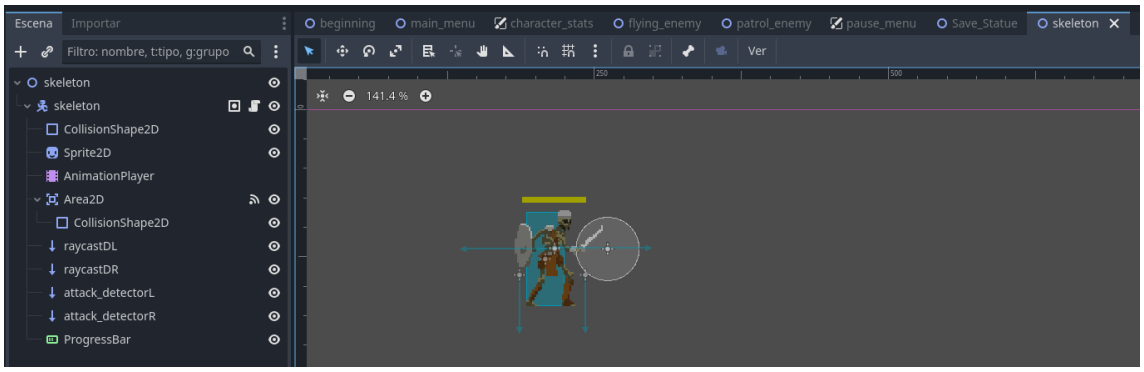


Ilustración 48. Escena del enemigo esqueleto. Fuente: Elaboración propia, 2024

Como se puede ver, los nodos empleados son bastante similares a los que se emplean con el enemigo volador, aunque observamos dos diferencias notables:

- En primer lugar, los nodos de tipo *RayCast2D* tienen funciones distintas. Seguimos teniendo dos nodos que inician la animación de ataque al detectar al jugador (*attack_detectorL y R*), pero ahora contamos con otros dos nodos (*raycastDL y raycastDR*) que permiten al enemigo detectar los precipicios y darse la vuelta para seguir patrullando sin caerse.
- Y, en segundo lugar, se ha sustituido el nodo de *AnimatedSprite* por un nodo *AnimatedSprite*, debido principalmente al formato en el que venían los recursos de imagen, que hacían que fuera mucho más sencillo de implementar las animaciones con este último tipo de nodo.

Analizando el script asociado a esta escena (*skeleton.gd*) observamos las siguientes funcionalidades:

- ***_ready***: se ejecuta al inicio, configurando las variables iniciales como la vida del esqueleto, su velocidad, y desactivando la colisión del área de ataque. Además, se configura la animación de "caminar" por defecto para el esqueleto.

- ***_physics_process***: se ejecuta en cada frame del juego y controla el movimiento del esqueleto. Detecta si el esqueleto está en contacto con una pared o al borde de una plataforma, cambiando la dirección del movimiento en consecuencia. También se encarga de actualizar la animación según la dirección en la que se mueve el esqueleto.
- ***detection***: determina si el esqueleto está en posición para atacar al jugador. Si el detector de ataque en la izquierda o la derecha colisiona con el jugador, el esqueleto se detiene y ejecuta un ataque.
- ***attack***: detiene el movimiento del esqueleto y ejecuta la animación de ataque. Una vez terminada la animación, el esqueleto reanuda su movimiento en la dirección en la que estaba originalmente.
- ***_player_body_detected***: se ejecuta cuando el área de ataque del esqueleto entra en contacto con el cuerpo del jugador. Si el jugador es detectado, se le inflige daño.
- ***hit***: se ejecuta cuando el esqueleto recibe daño. Reduce su barra de vida y, si la vida llega a cero, llama al método *dead*. Si no, muestra la animación de recibir daño y permite que el esqueleto continúe su funcionamiento normal.
- ***dead***: maneja la muerte y resurrección del esqueleto. Si el esqueleto tiene vidas restantes, reproduce una animación de muerte temporal y lo revive después de un breve retraso. Si no le quedan vidas, el esqueleto es eliminado del juego.
- ***set_life_color***: cambia el color de la barra de vida del esqueleto en función de las vidas que le quedan. El color cambia de verde a naranja y finalmente a rojo para indicar las etapas de vida del enemigo.

Enemigo terrestre 2: *PATROL_ENEMY*

Esta escena está asociada al enemigo con apariencia de goblin, cuyo comportamiento se basa en patrullar hasta que detecta al jugador, momento en el que se detiene para atacarle. Esta escena aparece instanciada en varias escenas, y su número de instancias depende exclusivamente del número de enemigos de este tipo que quieran incluirse en la escena. Los nodos que componen esta ventana son los que se pueden observar en la ilustración 49.

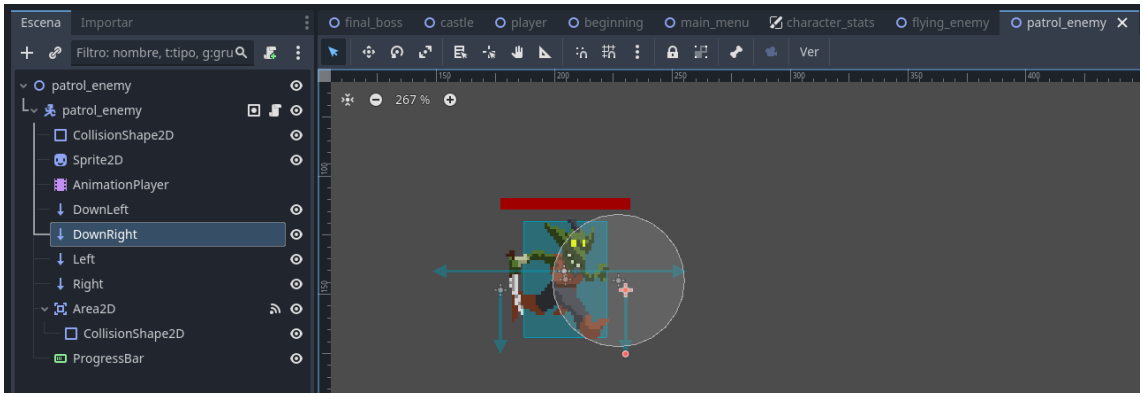


Ilustración 49. Escena del enemigo goblin. Fuente: Elaboración propia, 2024

Como se puede observar, la estructura de nodos es similar a la del primer enemigo terrestre, por lo que no se va a hacer hincapié en la utilidad de dichos nodos, ya que se ha explicado con la escena instanciable anterior.

Como consecuencia de la estructura, este tipo de enemigo también tiene un comportamiento prácticamente idéntico al de la escena anterior, con la diferencia de que este enemigo cuenta con más puntos de vida y únicamente es necesario abatirlo una sola vez para derrotarlo.

7.5.3. Estatua de guardado

Esta escena está asociada a la estatua con la que el jugador puede interactuar para guardar partida. Esta escena aparece instanciada en varias escenas, y su número de instancias depende del número de escenas donde aparezca, ya que únicamente se puede instanciar una vez por escena (para establecer un único punto de guardado en la escena). Los nodos que componen esta ventana (ver Ilustración 50) son los siguientes:

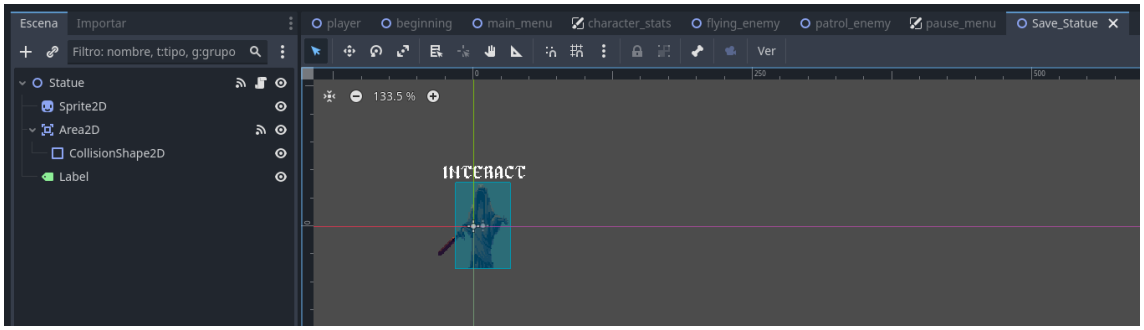


Ilustración 50. Escena de la estatua de guardado. Fuente: Elaboración propia, 2024

- *Sprite2D*: nodo que muestra el diseño de la estatua.
- *Area2D*: nodo que define el área de interacción del jugador con la estatua para guardar partida.
- *Label*: nodo que notifica al jugador cuando puede interactuar con la estatua, es decir, cuando se encuentra dentro del *Area2D*.

Analizando el script asociado a esta escena (*save_statue.gd*) observamos las siguientes funcionalidades:

- ***on_ready*:** se ejecuta al cargar la escena. En él, se establece que el *Label* no sea visible cuando se inicializa la escena.
- ***on_area_2d_body_shape_entered***: se activa cuando un cuerpo (en este caso, el jugador) entra en el área de colisión del nodo *Area2D* asociado a la estatua. Si el cuerpo que entra es el jugador, el *Label* se vuelve visible, indicando que el jugador puede interactuar con la estatua.
- ***on_area_2d_body_shape_exited***: se ejecuta cuando un cuerpo sale del área de colisión del *Area2D*. Si el cuerpo que sale es el jugador, el *Label* se oculta, indicando que ya no se puede interactuar con la estatua.

- ***input*:** Este método detecta si el jugador presiona la tecla asociada a la acción de interacción mientras está dentro del área de colisión de la estatua. Si se cumple esta condición, se llama al método *_save_game* para guardar la partida, y luego se ejecutan los métodos *loading_feedback* y *restore_life* del jugador, que proporcionan retroalimentación visual y restauran la salud del jugador.
- ***save_game*:** Este método gestiona el proceso de guardado del juego. Primero, verifica si existe un directorio de guardado y lo crea si no existe. Luego, abre o crea un archivo de guardado en ese directorio. Si el archivo no se puede abrir, muestra un mensaje de error. A continuación, obtiene todos los nodos en el árbol de la escena que están en el grupo "Persist" y llama al método *save* de cada uno para obtener los datos que deben ser guardados. Estos datos se convierten a una cadena JSON y se almacenan en el archivo de guardado.

7.6. Escenas jugables

A continuación, vamos a hablar de la implementación de las escenas jugables. En general, las tres escenas jugables comparten características de implementación muy similares. Por un lado, si analizamos los nodos podemos observar que tienen en común los siguientes:

- *Black:* conjunto de nodos empleado para la transición entre escenas.
- *TileMap* (uno o varios): nodo que genera un sistema de cuadrículas en la escena, empleado para construir el nivel mediante conjuntos de bloques (o *TileSets*) importados de una imagen. Un ejemplo de uso puede ser en la ilustración 56, dodnde a diferencia del fondo hecho con imágenes y un conjunto de nodos Parallax, el suelo y las casas están elaborados con este nodo. Además de permitir diseñar el nivel, permite determinar cómo funcionan las colisiones de dichos bloques, creando por ejemplo paredes, plataformas o suelos rígidos.
- *Conjunto Parallax*: conjunto empleado para anidar capas de fondo y dar una sensación de profundidad de la escena y movimiento del paisaje.
- Nodos *Marker2D*: nodos empleados para establecer los puntos de aparición del jugador, sea tras una transición de escena, una muerte o tras cargar partida.
- Nodos *Area2D* para los cambios de escena: nodo que detecta cuando el jugador se acerca a los límites de la escena, para así empezar a hacer el cambio de escenas.
- *AudioStreamPlayer*: nodo empleado para la reproducción de música en la escena.
- Escenas hijas instanciadas: corresponde principalmente a los enemigos de la escena jugable.

La escena correspondiente al pueblo (ver Ilustraciones 51 y 52) representa la estructura jugable más básica. Aquí se implementan los nodos mencionados anteriormente sin mayores añadidos, proporcionando un entorno sencillo en comparación con las demás escenas.

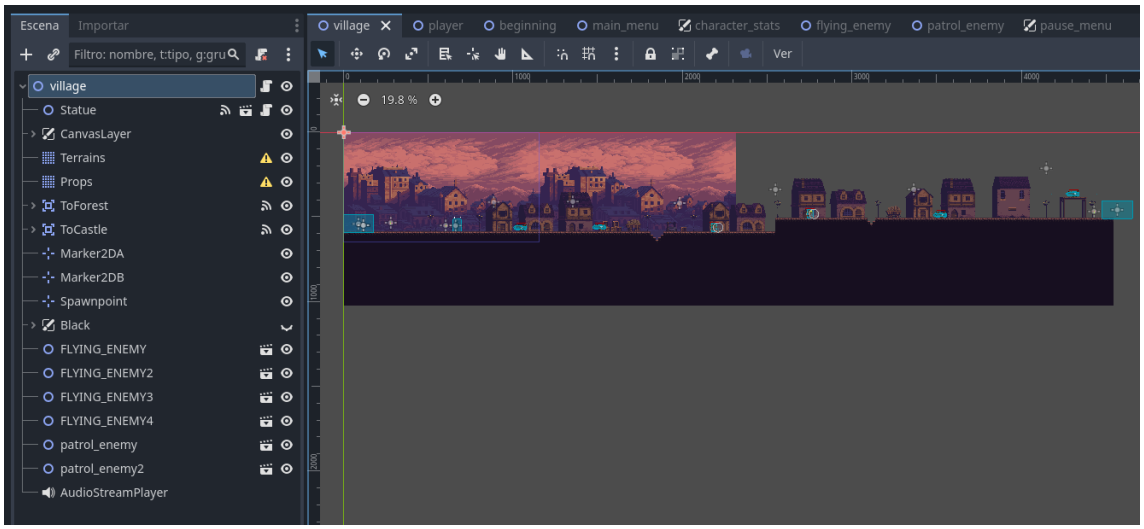


Ilustración 51. Escena de la zona del pueblo. Fuente: Elaboración propia, 2024

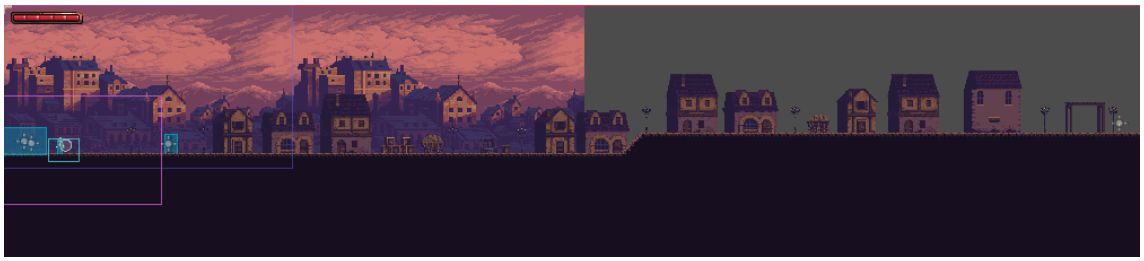


Ilustración 52. Diseño de la demo técnica para la sección del pueblo. Fuente: Elaboración propia, 2024

En la escena inicial (ver Ilustraciones 53 y 54), situada en la zona del bosque, se incrementa la complejidad al añadir nodos adicionales, como un *Area2D* extra y más *Marker2D*. Estos nodos adicionales se utilizan para implementar una trampa de pinchos, que penaliza al jugador con la pérdida de un punto de vida si cae en ella. En caso de sobrevivir, el jugador reaparecerá en uno de los extremos del tramo de pinchos, listo para intentar cruzar de nuevo. Si la pérdida de vida resulta en la muerte del personaje, se activan las correspondientes mecánicas de juego para manejar esta situación, tal como se define en el método ***_on_spikes_body_entered*** del script *ForestZone.gd*.

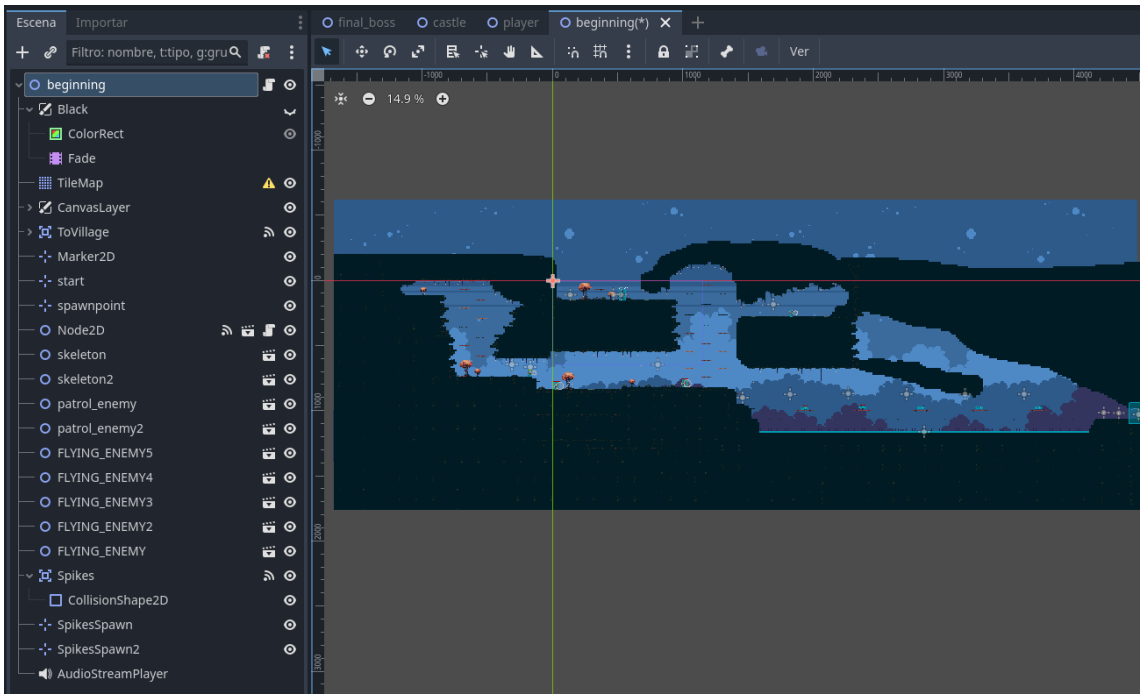


Ilustración 53. Escena de la zona del bosque. Fuente: Elaboración propia, 2024

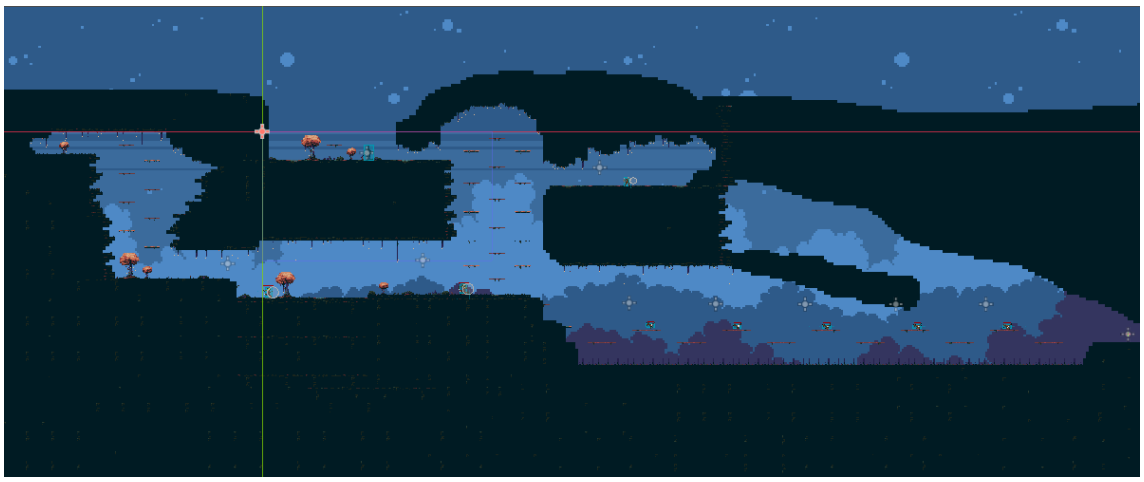


Ilustración 54. Diseño de la demo técnica para la sección del bosque. Fuente: Elaboración propia, 2024

Para terminar este incremento de funcionalidades en las escenas, tenemos la correspondiente a la zona del castillo (ver Ilustraciones 55 y 56). En esta escena, además de contar con las funcionalidades de las dos escenas anteriores, observamos dos adiciones más: por un lado se ha añadido un nodo de tipo *ColorRect* (*Hidden_Section*), encargado de ocultar una zona secreta hasta que se derrote al jefe del castillo; y en segundo lugar, un conjunto de nodos con un funcionamiento similar al de la estatua de guardado, aunque en este caso el diseño es el de un portal, el cual permite al jugador finalizar la versión de demostración del juego al cruzarlo.

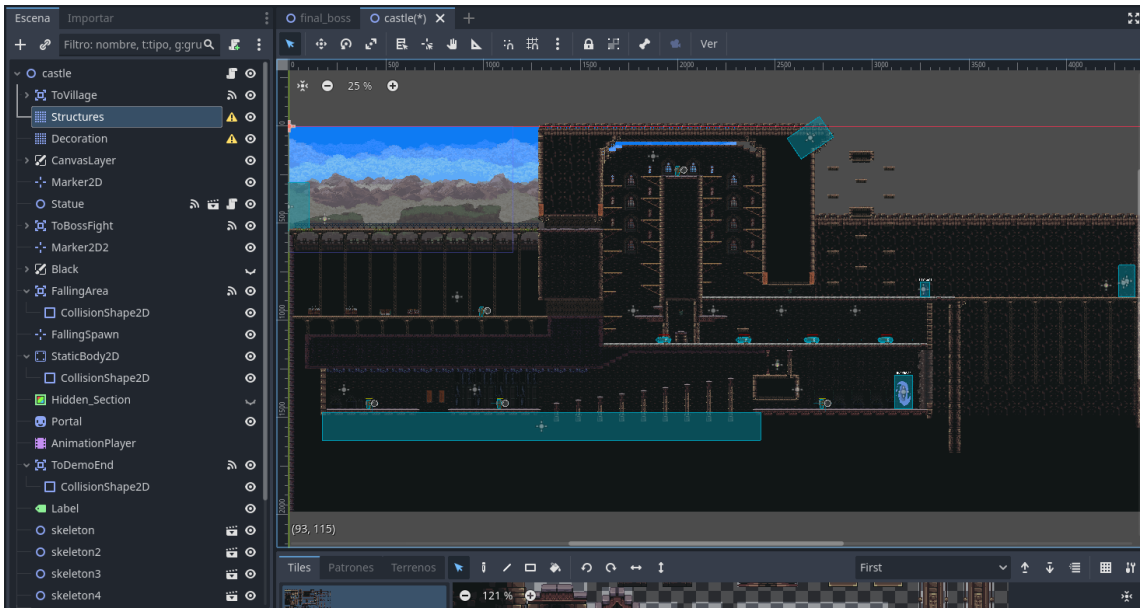


Ilustración 55. Escena de la zona del castillo. Fuente: Elaboración propia, 2024

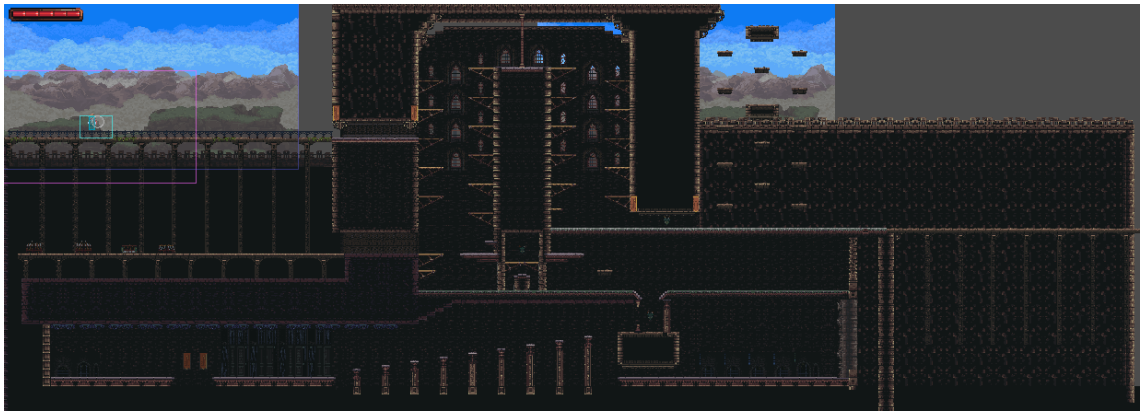


Ilustración 56. Diseño de la demo técnica para la sección del castillo. Fuente: Elaboración propia, 2024

En cuanto al resto de la implementación vía código, podemos destacar tres funcionalidades comunes principalmente:

- ***_ready***: establece varios atributos esenciales, como el estado del juego (por ejemplo, si es la primera carga, si se está cargando una partida guardada, o si el personaje está muerto). También configura la pantalla en modo de pantalla completa y carga el jugador dentro de la escena actual. De esta manera, cada vez que se inicia una nueva escena, se asegura que todos los parámetros necesarios estén correctamente configurados, lo que permite al jugador comenzar o continuar su partida sin problemas.
- ***_player_set_up***: se encarga de posicionar y escalar al personaje dentro de la escena, así como de configurar la cámara que sigue al jugador. Este método asegura que, dependiendo del estado del juego (por ejemplo, si es una nueva partida, una carga o una reaparición después de la muerte), el jugador aparezca en la posición correcta y la cámara esté correctamente alineada. Además, se gestiona la animación de fundido de la pantalla para suavizar la transición visual entre

diferentes partes del juego, lo que proporciona una experiencia de usuario más cohesiva y fluida.

- **métodos relacionados con la detección y gestión de transiciones entre escenas**, como por ejemplo ***_on_to_village_body_shape_entered***: detectan cuándo el personaje interactúa con ciertas áreas dentro de la escena (en este caso los límites de la escena) y desencadenan una secuencia de animaciones de transición, seguidas por el cambio de escena. Estos métodos son cruciales para mantener la continuidad narrativa y espacial dentro del juego, permitiendo al jugador moverse de manera coherente y sin interrupciones de una escena a otra. Además, registran el estado del jugador, como su vida actual, para asegurar que estos datos se mantengan consistentes a lo largo del juego.

7.7. Escena del combate final

La escena correspondiente al combate contra el jefe final (ver Ilustraciones 57 y 58) es la escena más peculiar de todo el proyecto. Aunque inicialmente comparte muchas similitudes con el resto de las escenas jugables (pues contiene los mismos elementos con la misma utilidad, como el *TileMap*, el *Marker2D* y la estatua instanciada), su comportamiento se ve condicionado por la implementación del jefe final, que dado que no se va a reutilizar en ningún momento se ha tomado la decisión de implementarlo en la misma escena. Por lo tanto, el comportamiento de esta escena comparte similitudes con los de las escenas jugables y con los de las escenas instanciables de los enemigos, especialmente con la escena del enemigo volador, ya que el jefe final no patrulla, sino que avanza hacia el jugador en el momento que lo detecta y, cuando entra en su rango de ataque, ejecuta dicha animación. Entre las características más distintivas de la escena, encontramos las siguientes:

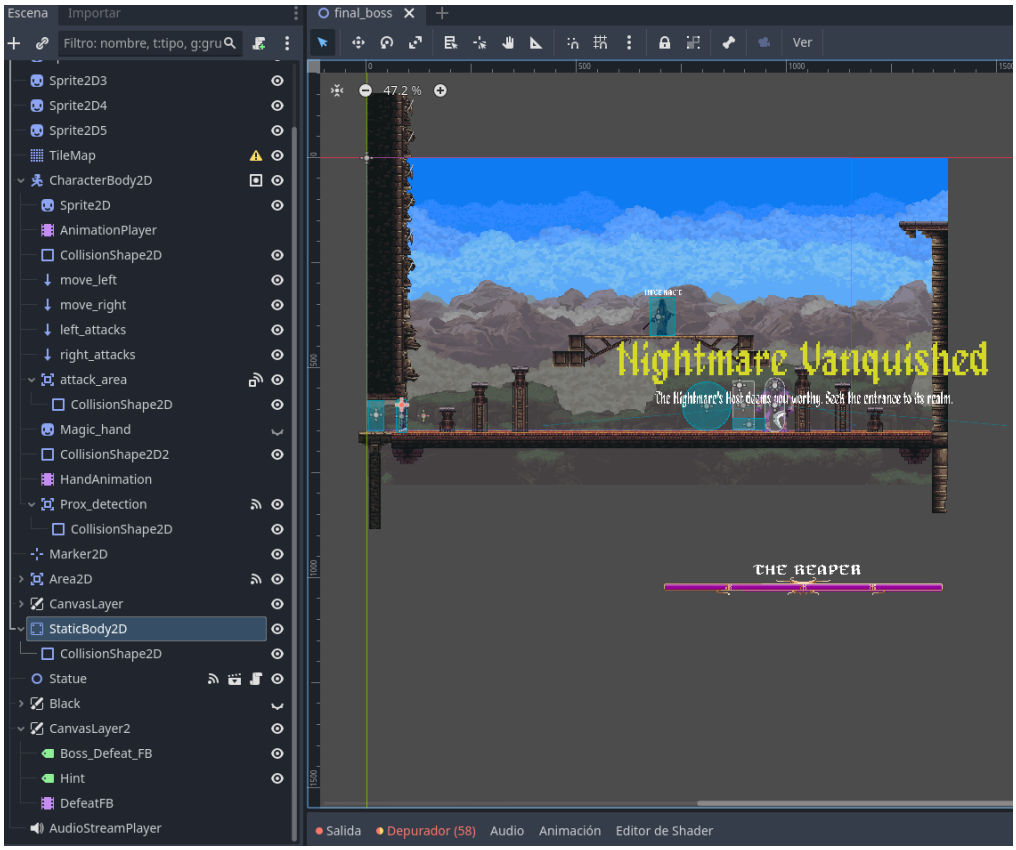


Ilustración 57. Escena de la zona del combate final. Fuente: Elaboración propia, 2024

Ilustración 58. Diseño de la demo técnica para la sección del combate contra el jefe. Fuente: Elaboración propia, 2024

- No hay Parallax: dado que el combate se realiza en una sala fija y no va a haber desplazamiento horizontal notable, se ha sustituido por un conjunto de nodos *Sprite2D* anidados para crear el paisaje de fondo.
- *Magic_hand*, *CollisionShape2D2*, *Hand Animation* y *Prox_detection*:
- *CanvasLayer*: contiene un conjunto de nodos que muestra de manera fija en la parte inferior de la pantalla la vida restante del jefe final.
- *StaticBody2D*: es un nodo que aplica una colisión en la rejilla de la esquina inferior izquierda de la escena, de manera que una vez empieza el combate, el jugador no puede abandonarlo hasta derrotarlo o ser derrotado. Esta rejilla y colisión desaparecen permanentemente una vez el jugador ha derrotado al jefe.
- *CanvasLayer2*: contiene un conjunto de nodos que muestran dos mensajes de retroalimentación tras derrotar al jefe final, indicando que se ha derrotado un enemigo poco común y cuál es la siguiente acción que debe realizar el jugador.

En cuanto al script asociado a esta escena (*final_boss.gd*) observamos las siguientes funcionalidades:

- ***_ready***: inicializa el entorno cuando la escena se carga por primera vez. Configura los elementos del jugador y del jefe final, como la cámara, la barra de salud y las posiciones iniciales, entre otros. Además, carga al jugador y se encarga de la configuración inicial de la escena en función de si el jefe del castillo ha sido derrotado previamente o no.
- ***_physics_process***: se ejecuta en cada frame y maneja la lógica de movimiento y ataque del jefe final. El jefe se mueve y ataca en función de las colisiones detectadas por los rayos y áreas de detección que se utilizan para determinar la proximidad y la posición del jugador.
- ***_play_animation***: se encarga de reproducir las animaciones correspondientes del jefe en función de si se está moviendo, atacando, o en reposo. Espera a que las animaciones de ataque o daño finalicen antes de permitir otros movimientos.
- ***_set_colliders_and_detectors***: ajusta la posición de los colisionadores y detectores en función de la dirección en la que mira el jefe (izquierda o derecha), asegurando que las colisiones se detecten correctamente según la animación.
- ***hit***: maneja el daño recibido por el jefe. Reduce la barra de vida y, si la vida llega a cero, ejecuta el método *death* para manejar la muerte del jefe. Si el jefe no muere, reproduce la animación de recibir daño.
- ***death***: maneja la secuencia de muerte del jefe, incluyendo la reproducción de la animación de muerte y la limpieza de ciertos elementos en la escena, como los colisionadores. También activa la estatua en la escena y reproduce una retroalimentación visual para indicar la derrota del jefe.
- ***fade_out_audio***: tras derrotar al jefe, se ejecuta y gradualmente disminuye el volumen de la música hasta silenciarla, y luego detiene la reproducción.
- ***_on_attack_area_body_entered***: se activa cuando el jugador entra en el área de ataque del jefe, causando daño al jugador.
- ***_on_area_2d_body_shape_entered***: se activa cuando el jugador entra en el área de transición, realizando así el cambio de escena hacia el castillo.

- ***_on_prox_detection_body_shape_entered***: se activa cuando el jugador entra en el área de detección de proximidad del jefe, haciendo que realice un ataque especial en el que ataca al jugador cerca de su propio cuerpo.

CAPÍTULO 8:

PRUEBAS

En este capítulo se presentan las pruebas realizadas sobre las diferentes partes del videojuego desarrolladas hasta el momento. Estas pruebas se han dividido en tres grandes bloques: en primer lugar, se evaluarán las escenas instanciables, que corresponden a aquellas escenas o componentes que pueden ser reutilizados en diferentes partes del juego. En segundo lugar, se analizarán las escenas no instanciables, que son aquellas que tienen una función única dentro del proyecto y no se replican. Por último, se expondrán las pruebas de usuario realizadas para validar la experiencia del jugador, obteniendo retroalimentación directa sobre la jugabilidad, accesibilidad y otros aspectos clave del juego.

8.1. Pruebas de escenas instanciables

En primer lugar, se presentan los resultados obtenidos tras realizar las pruebas con las escenas instanciables, que son aquellas escenas que pueden reutilizarse a lo largo del juego. Para poder realizar estas pruebas, se ha creado una escena estática (que no pertenece a ninguna de las zonas del mapa del videojuego) y se han instanciado los diferentes componentes que podían ser reutilizados, para poder comprobar mediante la interacción del usuario si las funcionalidades implementadas se ejecutaban correctamente. Los errores encontrados han sido los siguientes:

ESCENA INSTANCIABLE *player*

- Transparencia de la interfaz de muerte. En las primeras pruebas del juego, el fondo oscuro de la ventana que notificaba la muerte del jugador era completamente transparente debido a un error accidental en el editor del motor gráfico, donde se había ocultado inadvertidamente mientras se revisaba otra funcionalidad. Para evitar errores similares en el futuro, se decidió manejar la visibilidad de este fondo a través de código. De esta manera, el fondo permanece invisible al cargar la escena y se muestra únicamente cuando el personaje muere, asegurando que su comportamiento no se vea afectado por modificaciones en la vista de la escena.
- Aparición e interacción con los menús de pausa. Inicialmente, tanto el menú de pausa como el de estadísticas del personaje estaban integrados en la escena del jugador, permaneciendo ocultos hasta que se activaba el evento que alternaba su visibilidad. Sin embargo, surgió un problema con la prioridad de las señales emitidas por los nodos en Godot, donde la jerarquía de la escena determinaba qué menú respondía a las interacciones. Esto causaba que, si el menú de pausa estaba ubicado por encima del menú de estadísticas en la jerarquía, solo se detectaban las señales del primero, bloqueando el segundo. Para resolverlo, se optó por separar ambos menús en escenas independientes, eliminándolos de la escena del jugador. Ahora, en lugar de alternar la visibilidad de estos menús, se crean y destruyen instancias de las escenas correspondientes durante la navegación.
- Animación de recuperación de vida de la función *restore_life()*. El problema con la animación de recuperación de vida radicaba en que, independientemente de cuántos puntos de vida le faltaran al jugador, la animación siempre mostraba como si se recuperaran todos los puntos de vida al mismo tiempo. Para solucionar esto, se añadió un bloque condicional en la función que establece el fotograma inicial de la animación según los puntos de vida actuales. Por ejemplo, si el jugador tiene 4 de 5 puntos de vida, la animación solo mostrará la recuperación de un punto de vida.

- Reinicio de la barra de vida durante los cambios de escena. Durante los cambios de escena, la barra de vida se reiniciaba al máximo, sin importar cuántos puntos de vida le quedaban al jugador, porque la instancia de la escena A se eliminaba y se creaba una nueva para la escena B. Para solucionar este problema, se creó un atributo en el fichero general que almacena los puntos de vida de la escena anterior, permitiendo que el valor correcto se transfiera a la escena posterior. De esta manera, tanto el atributo de vida del jugador como el fotograma correcto de la barra de vida se configuran adecuadamente al iniciar la nueva escena.

ESCENA INSTANCIABLE *skeleton*

- Daño constante al jugador sin realizar animación. Este problema surgía porque, después de que el esqueleto fuera abatido por primera vez, su objeto de colisión asociado al ataque no se desactivaba, lo que causaba que el jugador recibiera daño constante al acercarse. Para corregir este error, se implementaron en el código una serie de sentencias que garantizan que el objeto de colisión se active únicamente durante la animación de ataque y se desactive al finalizar la misma, permaneciendo inactivo durante el resto de las acciones del enemigo.
- Cuerpo del personaje permanece deshabilitado. Este problema también aparecía después de que el enemigo fuera abatido por primera vez, ya que su cuerpo se deshabilitaba y no se reactivaba cuando el enemigo revivía. Para solucionarlo, se añadió una línea de código que reactiva el cuerpo del enemigo tras unos segundos, siempre que no haya sido abatido tres veces, asegurando que el enemigo funcione correctamente en revividas posteriores.

8.2. Pruebas de escenas no instanciables

A continuación, se presentan los resultados obtenidos tras realizar las pruebas con las escenas no instanciables, que son aquellas escenas que no pueden reutilizarse a lo largo del juego (como son las diferentes zonas del mapa jugable). Para poder realizar estas pruebas, se ha creado una versión jugable inicial correspondiente al contenido de la demo técnica y se ha lanzado a ejecución para comprobar que los nodos de la escena funcionaban correctamente. Los errores encontrados han sido los siguientes:

ERRORES GENÉRICOS DE TODAS LAS ESCENAS NO INSTANCIABLES

- Transiciones entre escenas. El principal problema en las transiciones entre escenas era la incorrecta ubicación de las coordenadas de reaparición. Al cambiar de una escena a otra, como se ilustra en la ilustración 59, si el jugador se movía del bloque derecho al izquierdo, aparecía en el círculo naranja en lugar del círculo negro correspondiente. Esto sucedía porque, aunque se había añadido un marcador mediante un nodo *Marker2D* para indicar el punto de reaparición, la lógica de transición entre escenas no estaba implementada correctamente. Para solucionar este problema, se incorporaron una serie de parámetros de posición (arriba, abajo, izquierda y derecha) al método de cambio de escena del archivo general. Estos parámetros son evaluados mediante un bloque condicional al cargar la nueva escena, determinando la posición exacta en la que debe aparecer el personaje.

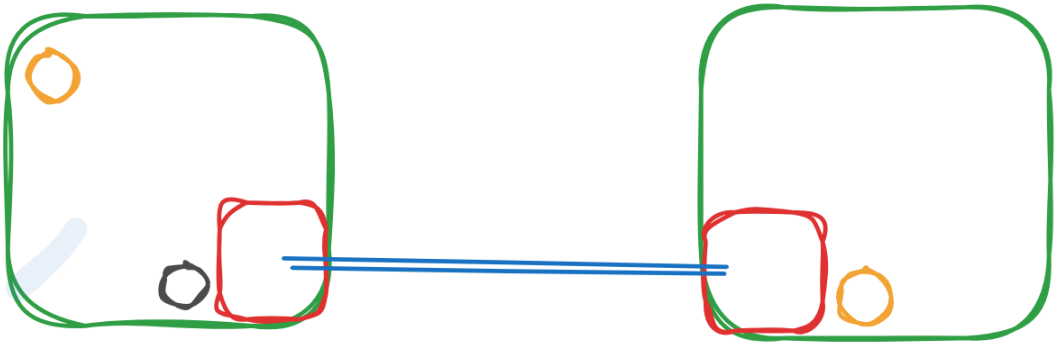


Ilustración 59. Esquema ilustrativo del error de transición entre escenas. Fuente: Elaboración propia, 2024

- Reaparición errónea al cargar partida. Al cargar una partida guardada, aunque se registraba correctamente la última escena visitada, el jugador reaparecía en el lado izquierdo de la escena en lugar de cerca de la estatua de guardado. Para corregir esto, se implementó un bloque condicional que distingue si la escena se está cargando debido a un cambio de escena o por la carga de un archivo guardado. Además, se establecieron las coordenadas de la estatua de guardado como el punto de reaparición al cargar la escena.

ESCENA NO INSTANCIABLE *beginning*

- Colisión de la trampa de pinchos. Durante la implementación, se detectó que la vida del jugador disminuía mucho antes de entrar en contacto con los pinchos de la trampa. Este problema menor se solucionó simplemente reajustando el área de colisión a través del editor gráfico, asegurando que el daño se aplique solo cuando el jugador efectivamente toque los pinchos.
- Problemas de reaparición en la trampa de pinchos. Este problema se originó debido a la resolución distinta de la escena *beginning*, que difiere de las demás. Como resultado, al utilizar las coordenadas de reaparición, se notaban diferencias significativas en la distancia en píxeles. Para corregirlo, se realizaron varias ejecuciones, ajustando la posición de reaparición mediante la adición o reducción de píxeles, hasta que el personaje reapareciera en el punto exacto deseado tras caer en los pinchos.

ESCENA NO INSTANCIABLE *village*

- Escaleras del pueblo. El problema principal radicaba en que el personaje no podía subir unas escaleras configuradas como un objeto de colisión inclinado, debido a que solo podía realizar movimientos horizontales al caminar. Inicialmente, se intentó solucionar este inconveniente implementando un área que, al detectar al personaje, le permitiera ejecutar un movimiento en diagonal, simulando así la subida de las escaleras. Sin embargo, dado que visualmente resultaba extraño debido a la falta de una animación específica para subir escaleras (los recursos de animaciones utilizados fueron

extraídos de internet), se decidió finalmente eliminar las escaleras y sustituirlas por un desnivel que el personaje pudiera superar mediante un salto.

ESCENA NO INSTANCIABLE *castle*

- Error en la transición entre el castillo y la zona de combate final. De manera similar al problema con la transparencia del menú de muerte, se produjo un error debido a un cambio temporal realizado en el editor gráfico, donde se había desactivado el área de colisión que permitía regresar de la zona de combate al castillo. Al igual que en el caso anterior, la solución consistió en gestionar la habilitación y deshabilitación de dicha área mediante código, asegurando así que no se vea afectada por modificaciones en el editor.

ESCENA NO INSTANCIABLE *final_boss*

- Punto de guardado del área del jefe final. El problema en esta escena era que, incluso después de guardar la partida tras derrotar al enemigo principal, el enemigo reaparecía al cargar la partida o cambiar de escena, cuando en realidad debería permanecer muerto. Para solucionar este problema, se añadió un atributo de punto de control (check_castle) que se almacena tanto en el archivo de guardado como en el fichero General. Este atributo persistente indica si el jefe de la demo ha sido derrotado, asegurando que no reaparezca.

8.3. Pruebas de usuario

Por último, y cerrando así la fase de pruebas, se presentan los resultados obtenidos tras realizar la prueba con un número reducido de usuarios. Para poder llevar a cabo esta prueba, y así comprobar que el videojuego también funciona correctamente en otros dispositivos, se ha exportado a un archivo ejecutable para diferentes sistemas operativos (en total 3: MacOS, Linux y Windows) con las correcciones de las dos primeras fases de prueba realizadas, y se ha enviado a una muestra de 18 jugadores, solicitándoles que jugaran de dos maneras diferentes: la primera partida, como una partida normal, simplemente progresando en la demo hasta completarla; y la segunda, tratando de realizar las pruebas que ellos consideraran oportunas con el objetivo de forzar cualquier posible error. En ambos casos, se les ha solicitado que documentaran los fallos, ya sea anotándolos junto con una pequeña descripción de que han hecho para provocar el error; o grabando la partida, ya sea completa o únicamente los fragmentos donde se ha producido un fallo. Las recomendaciones y errores encontrados han sido los siguientes:

- Animaciones del jugador no excluyentes cuando el jugador muere. Se ha añadido una línea de código que permite parar todas las animaciones del nodo *AnimationPlayer* antes de ejecutar la animación de muerte, para así evitar que se superpongan.
- Mecanismo de bloqueo de ataques del jugador. Los jugadores han incidido en que, aunque el bloqueo es funcional, desearían que pudiera realizarse un “bloqueo perfecto” para dinamizar más el combate, de manera que, si se ejecuta el bloqueo en un instante preciso del ataque del enemigo, este se quede aturdido y se pueda realizar un contraataque que haga mucho más daño que un ataque normal.

- Error en el efecto Parallax de varias escenas. Este error se debe a problemas en monitores con una resolución mayor de 1920 x 1080p. Este error se tendrá en cuenta para próximas iteraciones, de manera que las ventanas e interfaces sean *responsive*.
- Falta de objetos de colisión en el área de combate final. Se ha revisado el *TileMap* del combate final y se ha añadido área de colisión a los bloques que no la tenían, para así evitar que el personaje cayera por la derecha de la pantalla.
- Falta de objetos de colisión en el área del castillo. Se ha revisado el *TileMap* del castillo y se ha añadido área de colisión a los bloques que no la tenían, para así evitar que el personaje cayera por el puente ubicado en la zona inferior izquierda.
- Duración del mensaje de fin del combate excesivamente corta. Se ha añadido un temporizador que mantiene los mensajes de fin del combate durante unos segundos antes de desaparecer de la pantalla
- *Dash* infinito. Se ha añadido una variable de control correspondiente a un *cooldown*, de manera que si se hace un *dash* hay un tiempo de "enfriamiento" de esta animación, por lo que así no se puede realizar esta acción de manera infinita.
- *Boss final* fácilmente derrotable si el jugador se aproxima mucho. Se ha ampliado el área del ataque especial y se ha añadido un temporizador que repita este ataque si el jugador permanece mucho tiempo en las proximidades del jefe, forzando al jugador a tomar distancia para no recibir daño.
- Desaparición de la música. Solo se había configurado la opción de bucle de música en una de las escenas, así que se ha añadido al resto para que la música no detenga su reproducción.
- Detección de enemigos del jugador no funciona en ocasiones por la izquierda.
- Barra de vida y música del *boss final* no aparecen siempre. Se ha añadido la aparición de la música y la barra de vida también en el caso de que el jugador se aproxime al enemigo por la espalda.
- Además, durante las pruebas de usuario se han verificado todos los requisitos no funcionales establecidos, con la excepción del tercer requisito, el cual ha sido verificado parcialmente. Esto se debe a que todos los jugadores participantes contaban con sistemas que superaban con creces los requisitos de memoria y almacenamiento especificados. Sin embargo, algunos de los sistemas en los que se ejecutó la demo cumplían exactamente con los requisitos mínimos de sistema operativo y procesador, lo que permitió confirmar que el videojuego es ejecutable en estos dispositivos. En una próxima iteración se enfocarán los esfuerzos en verificar los dos últimos aspectos relacionados con memoria y almacenamiento.

CAPÍTULO 9:

PERSPECTIVA DE DESARROLLO Y CONCLUSIONES

9.1. Hitos alcanzados en el desarrollo del proyecto

El objetivo principal de este proyecto, que consistía en crear una primera versión jugable del videojuego con las mecánicas básicas implementadas, ha sido exitosamente alcanzado. Las 15 funcionalidades descritas en el apartado de alcance, esenciales para el núcleo de la jugabilidad, han sido desarrolladas y están plenamente operativas. Además, otro hito significativo es la creación de este documento, que no solo sirve como guía detallada para futuras iteraciones, sino que también facilita la comprensión del proyecto por parte de posibles colaboradores. Esta documentación proporciona una visión clara de la metodología utilizada, la disposición de los elementos del proyecto y su implementación técnica, lo que simplificará el mantenimiento y las mejoras en versiones futuras.

9.2. Aspectos por mejorar y posibles soluciones

Haciendo una retrospectiva de las fases de implementación y pruebas, se identifican cuatro áreas clave a mejorar:

- **Dinámica del combate**: Se propone mejorar la mecánica de bloqueo para hacer los combates más fluidos y estratégicos.
- **Adaptabilidad de las ventanas**: Es crucial que las interfaces del juego se adapten correctamente a diferentes resoluciones de pantalla, lo que también solucionaría los problemas detectados con el efecto Parallax durante las pruebas de usuario.
- **Movilidad del jugador**: Se sugiere ajustar el desplazamiento horizontal del personaje, implementando una aceleración gradual al iniciar y detener el movimiento, lo que añadiría realismo y control a la jugabilidad.
- **Estándares de codificación**: Implementar un estándar de buenas prácticas en la escritura de código mejoraría la consistencia, legibilidad y facilidad de depuración del proyecto en su conjunto.

9.3. Características para la siguiente iteración del proyecto

Considerando el alcance del proyecto y los hitos alcanzados en la primera fase, para la siguiente iteración del proyecto, se recomienda centrarse en las siguientes áreas:

- Mejoras técnicas propuestas: Aunque menos prioritarias, las mejoras sugeridas en el apartado anterior deberían ser implementadas para refinar la experiencia de juego.
- Nuevas funcionalidades: Es esencial agregar las funcionalidades 16, 17, 18, 20 y 23 del listado de alcance, que incluyen los sistemas de inventario, mejoras, coleccionables y zonas ocultas, para enriquecer la jugabilidad y ofrecer una experiencia más completa.
- Narrativa y diseño de niveles: Se debe profundizar en los principios de narrativa y diseño de niveles para hacer el juego más inmersivo y atractivo a largo plazo.
- Formación de un equipo de desarrollo: Es fundamental construir un equipo de desarrollo, priorizando la incorporación de un diseñador de artes conceptuales y un artista de pixel-art. Esto permitirá crear recursos visuales y de animación originales, aumentando el valor añadido del juego.

9.4. Qué se ha aprendido y cierre de la versión de demostración

A lo largo de este proyecto, se ha aprendido a elaborar una documentación integral para un proyecto de software, manejando los riesgos y desafíos que ello conlleva. Además, se ha adquirido un profundo entendimiento del motor gráfico utilizado, su lenguaje de programación y lógica interna. También se ha consolidado el conocimiento en programación orientada a objetos y eventos, aplicando de manera efectiva el patrón modelo-vista-controlador. Estas experiencias no solo han enriquecido las competencias técnicas, sino que también han fortalecido la capacidad de gestionar proyectos complejos en un entorno de desarrollo real.

BIBLIOGRAFÍA

Alessandrelli, F. (2022). The Godot editor running in a web browser. Recuperado de https://godotengine.org/article/godot-editor-running-web-browser/. (Fecha de consulta: 12/8/2024)

Aller, A. (2024). Motor gráfico: ¿Qué es? PCComponentes. Recuperado de https://www.pccomponentes.com/motor-grafico-que-es. (Fecha de consulta: 12/8/2024)

Andalucía Game. (2024). Juegos Soulslike: Cuando la dificultad es el atractivo. Andalucía Información. Recuperado de https://andaluciagame.andaluciainformacion.es/andalucia/1672174/juegos-soulslike-cuando-la-dificultad-es-el-atractivo/. (Fecha de consulta: 11/8/2024)

Cano, J. (2020). Análisis de Blasphemous [PC]. Vandal. Recuperado de https://vandal.elespanol.com/analisis/pc/blasphemous/48623#p-13. (Fecha de consulta: 12/8/2024)

Canle, E. (2023). Características de Unity, el motor gráfico más versátil del mercado. Tokio School. Recuperado de https://www.tokioschool.com/noticias/caracateristicas-unity/. (Fecha de consulta: 12/8/2024)

Carrión, R. (2022). Hollow Knight: las claves de su éxito. MeriStation. https://as.com/meristation/2022/05/21/reportajes/1653119710_756235.html. (Fecha de consulta: 11/8/2024)

Caurin, J. (2019). Videojuegos de plataformas | ¿Qué significa Videojuegos de plataformas? Geekno. https://www.geekno.com/glosario/videojuegos-de-plataformas. (Fecha de consulta: 11/8/2024)

Código Vanguardia. (s.f.-a). GameMaker Studio: Herramientas y características. Recuperado de https://codigovanguardia.com/desarrollo-de-videojuegos/gamemaker-studio-herramientas-y-caracteristicas/?utm_content=cmp-true. (Fecha de consulta: 12/8/2024)

Código Vanguardia. (s.f.-b). Godot Engine: Ventajas y características destacadas. Recuperado de https://codigovanguardia.com/desarrollo-de-videojuegos/godot-engine-ventajas-y-caracteristicas-destacadas/. (Fecha de consulta: 12/8/2024)

Devuego. (s.f.). Backtracking. Gamerdic. Recuperado de https://www.devuego.es/gamerdic/termino/backtracking/#:~:text=Del%20ingl%C3%A9s%20back%20tracking%20(recorrer,que%20previamente%20no%20eran%20accesibles. (Fecha de consulta: 11/8/2024)

DLCompare. (s. f.). Comprar y descargar Ori and the Blind Forest. https://www.dlcompare.es/juegos/100000061/comprar-descargar-ori-and-the-blind-forest. (Fecha de consulta: 12/8/2024)

Fernández, Y. (2023). Qué es Canva, cómo funciona y cómo usarlo para crear diseños. Xataka. https://www.xataka.com/basics/que-canva-como-funciona-como-usarlo-para-crear-diseno. (Fecha de consulta: 12/8/2024)

GitHub. (s.f.). Acerca de GitHub y Git. https://docs.github.com/es/get-started/start-your-journey/about-github-and-git. (Fecha de consulta: 12/8/2024)

Godot Engine. (s. f.). Saving games. Recuperado de https://docs.godotengine.org/es/4.x/tutorials/io/saving_games.html. Fecha de consulta: 18/8/2024)

Gómez, N. (2023). Unity: ¿Qué es y cómo se usa este motor gráfico en el diseño de videojuegos? El Tiempo. Recuperado de https://www.eltiempo.com/tecnosfera/videojuegos/unity-que-es-y-como-se-usa-este-motor-grafico-en-el-diseno-de-videojuegos-812172. (Fecha de consulta: 12/8/2024)

Hernández, R. (2021). El Modelo de Arquitectura MVC (Modelo-Vista-Controlador) explicado. freeCodeCamp. Recuperado de https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/. (Fecha de consulta: 18/8/2024)

Hobby Consolas. (2023). Análisis de The Last Faith para PS4, PS5, Xbox One, Xbox Series X|S, Nintendo Switch y PC. Recuperado de https://www.hobbyconsolas.com/reviews/analisis-last-faith-ps4-ps5-xbox-one-xbox-series-xs-nintendo-switch-pc-1335024. (Fecha de consulta: 11/8/2024)

Itch.io. (s.f.). About Itch.io. Itch.io. Recuperado de https://itch.io/docs/general/about. (Fecha de consulta: 12/8/2024)

Martínez, M. (2024). ¿Qué es la programación orientada a eventos? Recuperado de https://profile.es/blog/programacion-orientada-a-eventos/. (Fecha de consulta: 18/8/2024)

Microsoft. (s. f.). Uso del mando Xbox 360. Xbox Support. Recuperado de https://support.xbox.com/es-ES/help/xbox-360/accessories/controllers. (Fecha de consulta: 16/8/2024)

Nintendo. (2024). Hollow Knight para Nintendo Switch. https://www.nintendo.com/es-mx/store/products/hollow-knight-switch/. (Fecha de consulta: 11/8/2024)

Quesada, D. (2020). ¿Qué es un metroidvania? Aprende sobre videojuegos con Hobby Basics. Hobby Consolas. https://www.hobbyconsolas.com/reportajes/metroidvania-aprende-videojuegos-hobby-basics-603159. (Fecha de consulta: 11/8/2024)

ResearchGate. (s.f.). Figura 10. Modelo de proceso incremental. Recuperado de https://www.researchgate.net/figure/Figura-10-Modelo-de-proceso-incremental-Fuente_fig6_326571456. (Fecha de consulta: 11/8/2024)

Royhul, Z. (2023). Understanding Game Core Loop: A Comprehensive Guide. LinkedIn. Recuperado de https://www.linkedin.com/pulse/understanding-game-core-loop-comprehensive-guide-zakky-royhul-mun/. (Fecha de consulta: 18/8/2024)

Torres, M. (2024). Creando un Dark Souls 2D. Udemy. https://www.udemy.com/course/creando-un-dark-souls-2d/?couponCode=KEEPLEARNING. (Fecha de consulta: 11/8/2024)

Udemy. (s.f.). Acerca de Udemy. https://about.udemy.com/es/. (Fecha de consulta: 12/8/2024)

Universidad Europea. (2022). Programación orientada a objetos: Qué es y para qué sirve. Recuperado de https://universidadeuropea.com/blog/programacion-orientada-objetos/#:~:text=La%20programaci%C3%B3n%20orientada%20a%20objetos%20es%20un%20modelo%20de%20programaci%C3%B3n,l%C3%B3gica%20necesaria%20para%20esa%20manipulaci%C3%B3n. (Fecha de consulta: 18/8/2024)

Valve Corporation. (2024a). Blasphemous [Videojuego]. Steam. Recuperado de https://store.steampowered.com/app/774361/Blasphemous/?l=spanish. (Fecha de consulta: 12/8/2024)

Valve Corporation. (2024b). GameMaker. Steam. Recuperado de https://store.steampowered.com/app/1670460/GameMaker/?l=spanish

Vandal. (2023). Tales of Kenzera: ZAU es el nuevo metroidvania publicado por EA Originals y presenta su tráiler. Vandal. https://vandal.elespanol.com/noticia/1350767552/tales-of-kenzera-zau-es-el-nuevo-metroidvania-publicado-por-ea-originals-y-presenta-su-trailer/. (Fecha de consulta: 11/8/2024)

VidaExtra. (2015). Ori and the Blind Forest, análisis. VidaExtra. https://www.vidaextra.com/analisis/ori-and-the-blind-forest-analisis. (Fecha de consulta: 12/8/2024)

ANEXOS

Anexo I. RELACIÓN DEL TRABAJO CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE DE LA AGENDA 2030

GRADO DE RELACIÓN DEL TRABAJO CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE				
ODS	**Alto**	**Medio**	**Bajo**	**No procede**
1. Fin de la pobreza				X
2. Hambre cero				X
3. Salud y bienestar				X
4. Educación de calidad		X		
5. Igualdad de género				X
6. Agua limpia y saneamiento				X
7. Energía asequible y no contaminante				X
8. Trabajo decente y crecimiento económico				X
9. Industria, innovación e infraestructuras				X
10. Reducción de las desigualdades				X
11. Ciudades y comunidades sostenibles				X
12. Producción y consumo responsables				X
13. Acción por el clima				X
14. Vida submarina				X
15. Vida de ecosistemas terrestres				X
16. Paz, justicia e instituciones sólidas				X
17. Alianzas para lograr objetivos				X

Descripción de la alineación del TFG/TFM con los ODS con un grado de relación más alto.

El videojuego *Oniros: el reino de los sueños*, enfocado en la salud mental y la evolución de la mente de un niño a través de las cinco fases de la pérdida, se alinea estrechamente con el Objetivo de Desarrollo Sostenible (ODS) 4: "Educación de calidad". Este ODS busca garantizar una educación inclusiva, equitativa y de calidad, promoviendo oportunidades de aprendizaje durante toda la vida para todos.

En primer lugar, el videojuego actúa como una herramienta educativa innovadora, sensibilizando a los jugadores sobre la importancia de la salud mental y los procesos de duelo. A través de su narrativa, *Oniros* educa a los jugadores sobre las etapas emocionales que se experimentan tras una pérdida, contribuyendo así a una mayor comprensión y empatía hacia quienes atraviesan situaciones similares. Esto se alinea con la meta 4.7 del ODS 4, que promueve la educación para el desarrollo sostenible, incluyendo la promoción de una cultura de paz y no violencia, y la apreciación de la diversidad cultural.

En segundo lugar, la inclusión de elementos interactivos y educativos en el juego puede fomentar habilidades blandas y competencias emocionales en los jugadores, especialmente en un público joven. Esto respalda la meta 4.4 del ODS 4, que busca aumentar el número de jóvenes y adultos con competencias técnicas y profesionales relevantes para el empleo, el trabajo decente y el emprendimiento. A través del juego, los usuarios desarrollan habilidades como la resiliencia, la gestión emocional y el pensamiento crítico, preparándolos mejor para enfrentar desafíos tanto personales como profesionales.

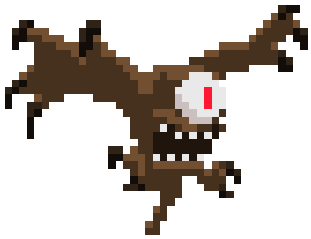
Finalmente, la accesibilidad del videojuego, disponible para diferentes plataformas y de manera gratuita, asegura que un amplio espectro de jugadores pueda beneficiarse de sus contenidos educativos. Esto promueve la equidad en el acceso a recursos educativos de calidad, en consonancia con la meta 4.1 del ODS 4, que se centra en garantizar que todos los jóvenes completen una educación primaria y secundaria gratuita, equitativa y de calidad con resultados de aprendizaje pertinentes y efectivos.

Anexo II. Diseños extraídos de internet para el desarrollo del videojuego.

En este anexo se muestran los diseños empleados para el desarrollo del proyecto, con el objetivo de mostrar de donde han sido extraídos.

- Diseño del personaje jugable extraído de: https://rvros.itch.io/animated-pixel-hero

- Diseño del enemigo volador extraído de: https://luizmelo.itch.io/monsters-creatures-fantasy

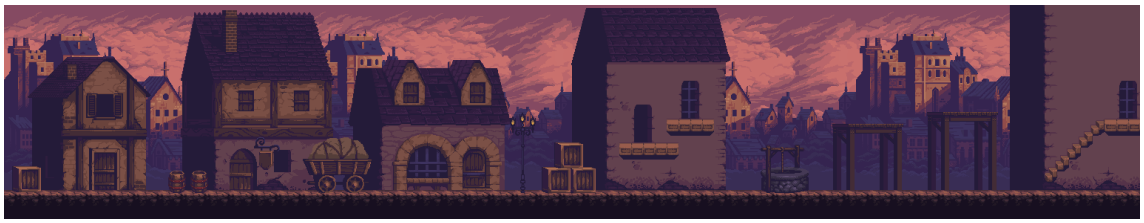


- Diseño del enemigo goblin extraído de: https://luizmelo.itch.io/monsters-creatures-fantasy

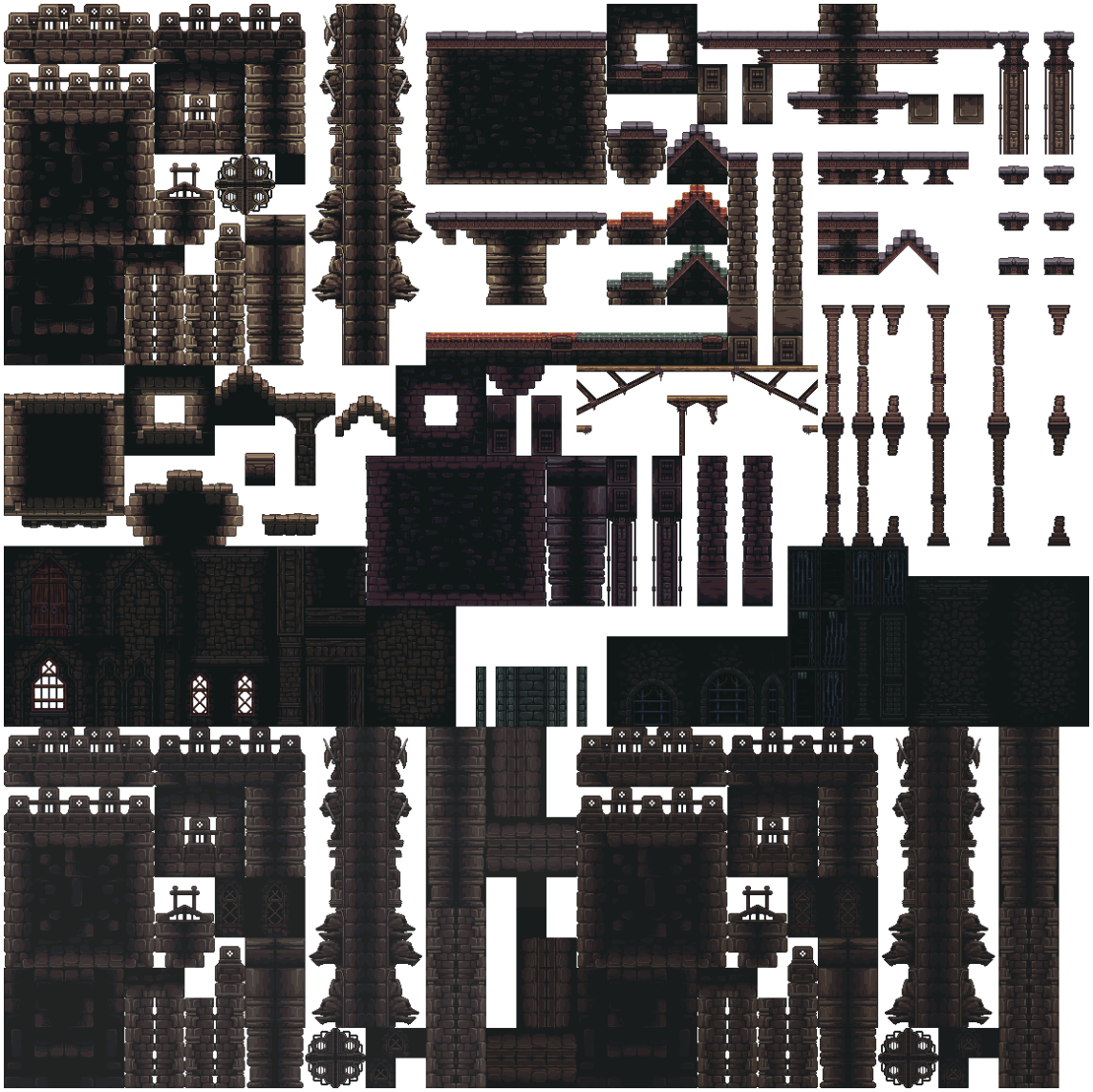
- Diseño del enemigo esqueleto extraído de: https://luizmelo.itch.io/monsters-creatures-fantasy

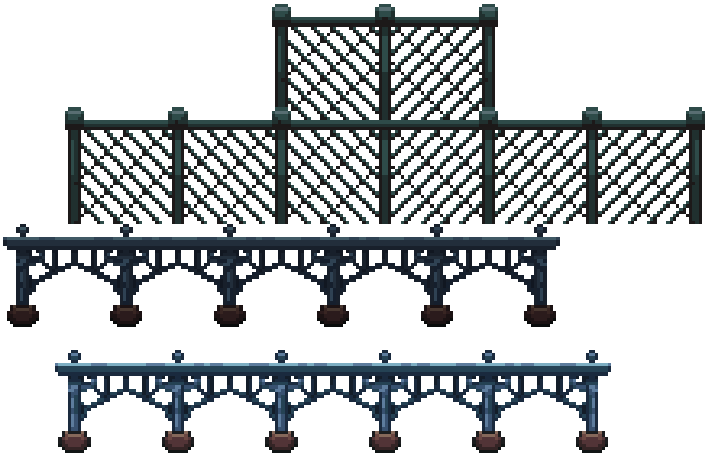
- Diseño de la zona del bosque extraído de: https://trixelized.itch.io/starstring-fields

- Diseño de la zona del pueblo extraído de: https://ansimuz.itch.io/gothicvania-town



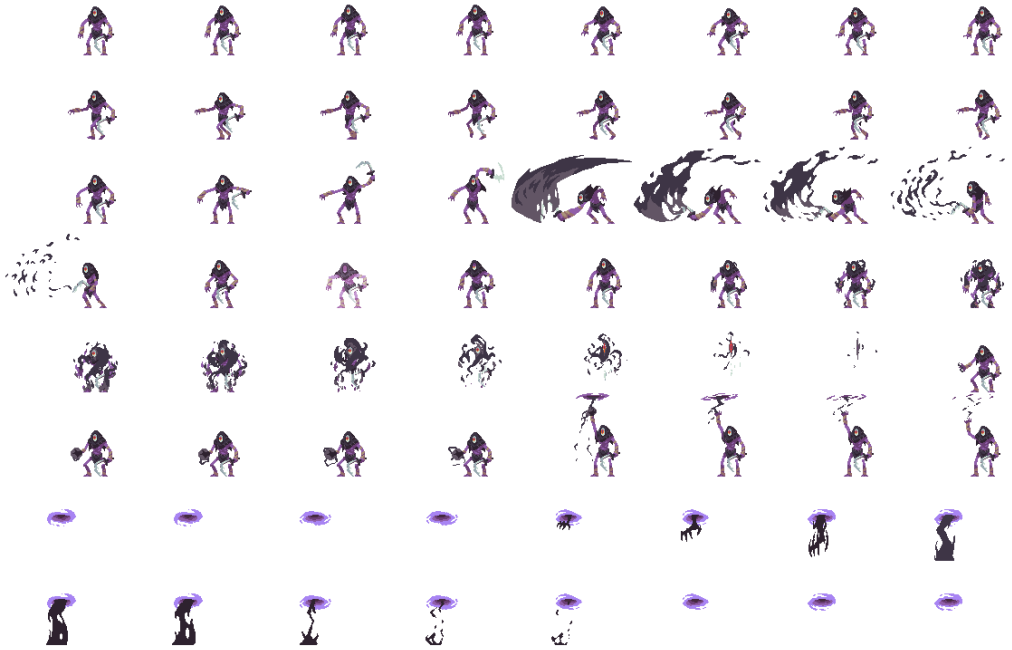
- Diseño de la zona del catillo extraído de: https://szadiart.itch.io/sidescroll-worlds-castle-pack



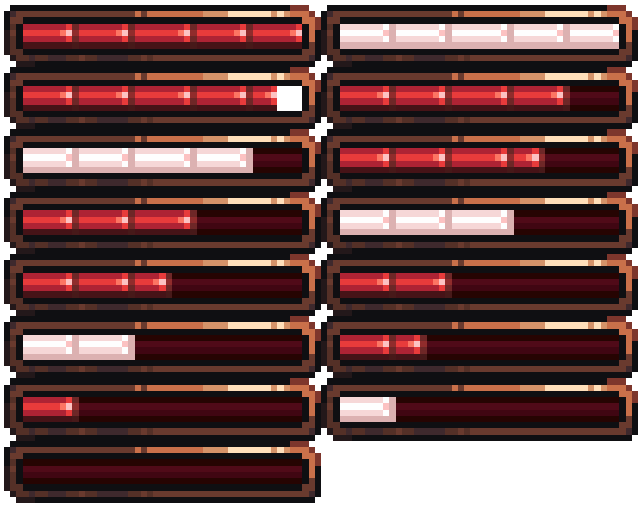


- Diseño del perro extraído de: https://penusbmic.itch.io/the-dark-series-dog-companion

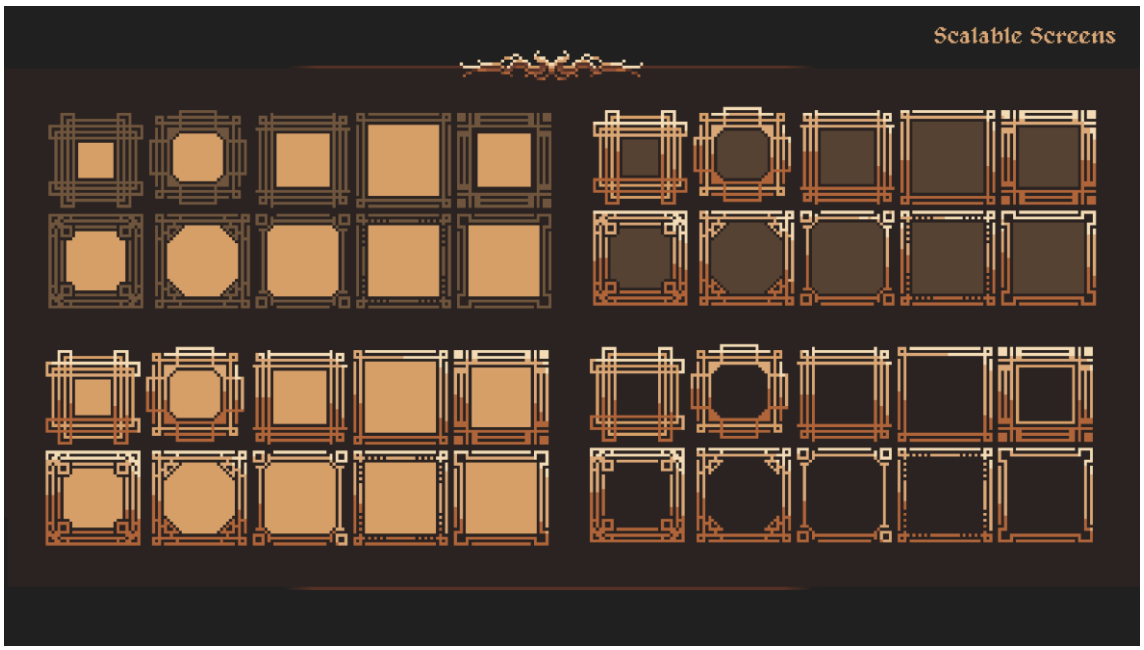
- Diseño de boss final extraído de: https://clembod.itch.io/bringer-of-death-free



- Diseño de barra de vida extraído de: https://fishxels.itch.io/hearts-n-bars



- Diseño de elementos de UI y HUD extraídos de: https://finnmercury.itch.io/ultilmate-dark-fantasy-ui-set



- Fuente para texto *Alagard* extraída de: https://www.dafont.com/alagard.font

Alagard