UNIVERSIDAD
POLITÉCNICA
DE MADRID

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Trabajo Fin de Grado

Videojuego de género roguelike 2D con carácter supervivencia desarrollado en Unity para PC

Grado Ingeniería de Software

Curso 2024 - 2025

Autor: Minwang Lou

Tutor: Javier Alcalá Casado

Madrid, 3 de julio de 2025

Agradecimiento

Quiero dar las gracias a todas las personas que me han ayudado de alguna manera durante el desarrollo de este proyecto.

Agradezco a mi tutor por validarme la idea principal que he tenido para este proyecto y por el apoyo posterior que me permitió llegar hasta el final del proyecto con éxito.

Agradezco a UPM por darme la oportunidad de hacer este proyecto y por todo lo que he aprendido en estos años. Ha sido una etapa que me permitió conocer, aprender y que seguro me servirá para mi camino en el futuro.

Además, quiero dar las gracias de forma muy especial a una amiga mía JialeWang, que me apoyó desde el momento en que todavía no tenía claro qué tipo de juego quería desarrollar para este proyecto. Más adelante, cuando pasé por una situación difícil que casi me impide graduarme este año, fue ella quien me apoyó y me ayudó a salir adelante. También en uno de mis peores días, cuando me sentía completamente bloqueada y no podía concentrarme nada en el desarrollo, después de hablar con ella, recuperé de nuevo la motivación de seguir adelante. Además, se participó en la prueba final del juego. Sin su apoyo, este proyecto seguramente no habría llegado hasta donde está ahora.

Por último, gracias a mí misma por llegar hasta aquí. Ha sido un recorrido largo y lleno de aprendizajes, no todo salió como esperaba, pero todo sirvió para avanzar, entender y estar preparado para lo que venga después.

Resumen

Este proyecto tiene como objetivo el desarrollo completo de un videojuego de tipo roguelike 2D, realizado de forma individual como trabajo final de grado para mi carrera de estudio. Desde el principio, la intención no fue simplemente crear un producto jugable, sino afrontar un proyecto real desde cero, en el que puedo aplicar todo lo que he aprendido durante la carrera y enfrentar todos los retos que implica construir algo funcional y coherente por cuenta propia.

Elegí desarrollar un videojuego porque es un tipo de proyecto que combina múltiples áreas: programación, diseño visual, estructura lógica, control de datos y experiencia de usuario. Todo eso lo convierte en una opción ideal para cerrar una etapa académica y poner a prueba lo aprendido en un entorno práctico y exigente.

El desarrollo del juego también refleja un interés personal por los juegos que ofrecen rejugabilidad, decisiones tácticas y mecánicas accesibles. Por eso, este proyecto se ha enfocado en crear una experiencia dinámica y flexible, donde cada partida sea distinta y el jugador tenga espacio para experimentar y mejorar.

A lo largo del documento se presenta el proceso completo del desarrollo: desde la motivación inicial hasta el resultado final, pasando por las decisiones de diseño, las herramientas utilizadas, los problemas enfrentados y las posibles ampliaciones futuras. Más allá del resultado final, este proyecto representa un aprendizaje completo, tanto a nivel técnico como personal.

Abstract

This project aims to develop a complete 2D Roguelike video game, carried out individually as a final degree project for my academic program. From the beginning, the goal was not just to create a playable product, but to face a real project from scratch — one in which I could apply everything I have learned throughout my studies and take on the challenges involved in building something functional and coherent on my own.

I chose to develop a video game because it is a type of project that combines multiple areas: programming, visual design, logical structure, data handling, and user experience. All of this makes it an ideal option to conclude an academic stage and put my knowledge to the test in a practical and demanding environment.

The development of the game also reflects a personal interest in games that offer replayability, tactical decision-making, and accessible mechanics. For this reason, the project has focused on creating a dynamic and flexible experience, where each playthrough feels different and the player is encouraged to experiment and improve.

This document presents the full development process: from the initial motivation to the final result, including design decisions, tools used, challenges encountered, and future improvements. Beyond the final product, this project represents a complete learning experience, both technically and personally.

Índice general

Índice de Figuras

Capítulo 1. Introducción

Desde que Pong llevó la diversión interactiva a los hogares en la década de 1970, los videojuegos han evolucionado de simples mecánicas de rebote a mundos complejos capaces de retar, sorprender y emocionar. Con cada salto tecnológico surgieron nuevas formas de enganchar al jugador: la gratificación inmediata de los arcades, la exploración abierta de las aventuras de 8 y 16 bits o la profundidad estratégica de los RPG clásicos.

Dentro de esta evolución constante han surgido géneros que, sin depender de grandes recursos técnicos, han sabido captar la atención de millones de jugadores. Uno de los más representativos es el roguelike, nacido con la aparición de Rogue en los años 80. Este tipo de juegos se caracteriza por niveles generados de manera procedimental, muerte permanente del personaje y partidas breves pero intensas. Esta combinación única crea experiencias irrepetibles, aumenta la tensión al enfatizar riesgos y recompensas, y ofrece ciclos cortos de juego que motivan constantemente al jugador a mejorar y volver a intentarlo. Gracias a esta fórmula, los roguelike se han consolidado entre los jugadores, gozando de gran popularidad especialmente en el ámbito independiente por su accesibilidad, rejugabilidad y sensación constante de progreso.

Magic Overdrive nace precisamente motivado por esta búsqueda de diversión atemporal y la exploración de mecánicas capaces de mantener el interés del jugador a largo plazo. Su propuesta combina la esencia clásica del roguelike con un estilo visual basado en pixel art retro, logrando una accesibilidad inmediata mediante controles sencillos y directos. Al mismo tiempo, introduce capas adicionales de profundidad estratégica con elementos como la selección táctica de armas, la evolución progresiva de habilidades y un controlado sistema de aleatoriedad, generando situaciones variadas y estimulando constantemente al jugador a descubrir nuevas combinaciones y estrategias.

1.1. Motivación

La idea de este proyecto nació de las ganas de crear un videojuego propio, desde cero, de forma independiente. Siempre me ha interesado el desarrollo de un videojuego, y sentía que este era el momento ideal para empezar por primera vez a algo completo: no solo escribir código, sino también tomar decisiones de diseño, organizar mecánicas, gestionar recursos y asegurar que todo funcione como un producto real.

Después de repasar muchos de los juegos que he jugado a lo largo de los años y de analizar qué tipo de experiencia sería capaz de desarrollar con los medios disponibles, decidí hacer un juego de tipo roguelike en 2D. Este género, aunque en apariencia se ve que es simple en la jugabilidad, pero permite al mismo tiempo crear partidas dinámicas y que presenta con un alto nivel de profundidad que refuerza la diversión del juego.

Además, encajaba muy bien con el tipo de proyecto que podía desarrollar por mi cuenta: con un diseño modular, sin necesidad de una narrativa compleja ni animaciones pesadas, pero con muchas posibilidades de crecimiento y expansión.

Así surgió **Magic Overdrive**, una mezcla de ideas que ya me gustaban como jugador, y en mismo tiempo quería intentar como desarrollador. El proyecto me permitió aplicar muchas de las cosas que había aprendido durante la carrera: programación orientada a objetos, trabajo con estructuras de datos, diseño de sistemas reutilizables y gestión de una base de datos externa.

Más allá de lo técnico, este proyecto también fue una forma de ponerme a prueba. Hubo momentos en los que no sabía si iba a poder terminarlo, en los que todo parecía más grande que yo pensaba. Pero poco a poco, con planificación y cada paso que he pisado en el camino, logré avanzar hasta construir algo jugable, entretenible y realizado por sí mismo. Esa es, al final, la mayor motivación de todas.

1.2. Objetivos

El objetivo general del proyecto es desarrollar un videojuego roguelike en 2D funcional y completo, centrado en la experiencia del jugador, la variedad de mecánicas y la extensibilidad del sistema. Para ello, se plantean los siguientes objetivos concretos y alcanzables dentro del desarrollo del proyecto:

1. Diseño y estructura del juego

- Definir la arquitectura general del juego basada en principios de extensibilidad y reutilización.
- Implementar un sistema de gestión de escenas (pantalla de inicio, partida, selección de personaje/mapa, etc.).
- Gestionar el flujo del juego: inicio de partida, oleadas, pausas, final de partida, etc.

2. Jugabilidad y mecánicas principales

- Desarrollar el movimiento controlado del personaje por parte del jugador, así como su interacción con el entorno.
- Diseñar e implementar distintas armas con comportamientos únicos.
- Desarrollar un sistema de ataques automáticos que varíen según el arma equipada.
- Diseñar diferentes tipos de enemigos con comportamientos, atributos y gráficas variados.
- Implementar un sistema de oleadas con dificultad creciente según el paso de tiempo durante un partido.

3. Gestión de recursos, datos y rendimiento

- Diseñar e implementar un sistema de Object Pool para optimizar el rendimiento y gestionar eficientemente los objetos (bonus y armas) que se proporcionará al jugador cuando tiene recogido suficiente cantidad de experiencia y preparada para subirse de nivel.
- Implementar un sistema de guardado permanente para almacenar el progreso del jugador (bonificaciones adquiridas, desbloqueo de armas, etc.) en directorio permanente y en formato de Json.

- Diseñar y desarrollar diversos controladores específicos para la gestión de armas, mapas, atributos, etc.
- Cargar y transformar datos desde documentos en formato JSON, facilitando la configuración externa y la escalabilidad del proyecto.
- Implementar un sistema de generación dinámica de mapas infinitos, basado en la posición donde tiene localizado el jugador, junto con un controlador centralizado para gestionar todas aquellas funcionalidades que espera tener cuando se genera los bloques de mapas.
- crear un sistema de gestión de sprites que abarque almacenamiento, referencia en código y representación en escena de todos los recursos gráficos (UI, armas, iconos, personajes y enemigos).

4. Sistema de progreso y recompensas

- Diseñar un sistema de orbes elementales recolectables que sirvan para subir el nivel del personaje que ofrecerá al jugador posibilidad de seleccionar las mejoras de las armas y activar evoluciones.
- Implementar el sistema de evolución de armas.
- Diseñar bonificaciones ("bonus") clasificadas en dos tipos:
    - Bonificaciones obtenidas durante la partida (mejoras temporales).
    - Bonificaciones obtenidas fuera de la partida (mejoras permanentes adquiribles).
- Calcular el efecto de las bonificaciones sobre los atributos del personaje, armas y reflejarlas en el juego.

5. Interfaz de usuario (UI)

- Crear interfaces gráficas que permitan al jugador:
    - Al seleccionar un personaje o un mapa, la interfaz muestra una descripción detallada del elemento seleccionado, permitiendo al jugador conocer sus características antes de comenzar la partida.
    - Se implementa un sistema de navegación entre paneles mediante botones o acciones del usuario, que permite cambiar de pantalla, como desde el menú principal a la selección de personaje, y luego la selección del mapa para comenzar el partido.

- Las acciones del jugador se reflejan visualmente en la interfaz, como la selección de una bonificación o la confirmación de una compra, proporcionando feedback inmediato.
- La interacción con la interfaz está diseñada de forma intuitiva, mostrando claramente qué acción va a realizar el jugador y presentando el resultado esperado de forma inmediata y comprensible.

6. Gestión de audio y efectos de sonido

- Desarrollar un **Audio Manager** centralizado que controle todos los clips, canales y volúmenes del juego.
- Integrar **música de fondo** con transiciones y cambios dinámicos según fase de partida (menú, oleadas intensas, pantalla de victoria/derrota).
- Añadir **efectos de interfaz** (clic de botón, navegación entre paneles) con prioridad baja y mezcla automática sobre la música.
- Asignar **efectos de sonido contextuales**:
  - Recogida de objeto / orbe elemental.
  - Ataque de arma y colisiones con enemigos.
  - Daño recibido, estado crítico y muerte del personaje.
  - Subida de nivel junto con la aparición de Upgrate Selection Panel

1.3. Herramientas y tecnologías utilizadas

El desarrollo del juego ha requerido el uso combinado de distintas herramientas y tecnologías, seleccionadas por su compatibilidad, facilidad de integración y eficiencia dentro de un entorno de desarrollo independiente. Cada componente ha cumplido un papel específico, ya sea en la programación del juego, el diseño visual, la organización del proyecto o la colaboración.

En este apartado se detallan las herramientas principales utilizadas a lo largo del proceso, así como el criterio seguido para incorporarlas al flujo de trabajo.

1.3.1. Unity

Ha sido el motor principal sobre el que se ha construido todo el proyecto. Su orientación a proyectos en 2D, su interfaz modular y la gran comunidad de soporte que lo rodea lo convierten en una opción especialmente adecuada para desarrolladores independientes. En el juego, Unity se ha utilizado para gestionar escenas, físicas básicas, animaciones, partículas, colisiones y la lógica de juego en general.

Una de sus principales ventajas ha sido la posibilidad de trabajar de forma estructurada mediante prefabs, scripts reutilizables y sistemas de componentes. Esto ha facilitado la organización del proyecto, la implementación de mecánicas complejas sin necesidad de partir desde cero, y la posterior escalabilidad de funcionalidades. Además, su compatibilidad con C# ha permitido aplicar principios de programación orientada a objetos de forma clara y mantenible.

1.3.2. Visual Studio Code

Ha sido el entorno de desarrollo utilizado para escribir y organizar todo el código del proyecto. Su ligereza, capacidad de personalización y amplia variedad de extensiones lo convierten en una herramienta muy útil, especialmente en proyectos donde se requiere trabajar con diferentes archivos y formatos.

Durante el periodo de desarrollo se ha utilizado principalmente para programar en C#, editar archivos JSON y navegar cómodamente entre scripts y recursos.

1.3.3. GitHub

GitHub ha sido la plataforma empleada para el control de versiones y el almacenamiento del proyecto en la nube. Utilizando Git como sistema de gestión, se ha podido llevar un seguimiento ordenado de los cambios realizados, crear copias de seguridad de forma constante y revertir errores cuando ha sido necesario.

Aunque el desarrollo del proyecto ha sido individual, contar con un sistema de versionado ha resultado útil para mantener una estructura limpia, probar funcionalidades sin comprometer la versión estable y documentar el progreso a lo largo del tiempo. Además, al estar alojado en remoto, el repositorio puede servir como respaldo y facilitar futuras colaboraciones o revisiones externas si el proyecto se amplía en el futuro.

1.3.4. Aseprite

Aseprite ha sido la herramienta utilizada para trabajar con los elementos visuales del juego en formato pixel art. Su interfaz está diseñada específicamente para la edición de gráficos de baja resolución, lo que la convierte en una opción ideal para proyectos con estética retro.

Durante el desarrollo Aseprite se ha empleado principalmente para realizar ajustes y adaptaciones sobre recursos gráficos ya existentes. Esto ha incluido modificaciones menores, corrección de detalles y adaptación de sprites para integrarlos de forma coherente dentro del estilo visual del juego. Gracias a su enfoque directo y funcional, ha permitido trabajar con precisión sin requerir un proceso de creación desde cero.

1.3.5. Newtonsoft.Json

Newtonsoft.Json ha sido la biblioteca utilizada para manejar la lectura, escritura y conversión de archivos JSON dentro del proyecto. Gracias a su integración sencilla con C#, ha permitido estructurar datos de forma clara y reutilizable, facilitando la carga de configuraciones externas o la generación de archivos desde código.

Durante el desarrollo del juego, se ha utilizado para importar datos de forma dinámica, como estadísticas de personajes o propiedades de armas y también

para transformar datos en tiempo de ejecución en archivos JSON válidos. Esta flexibilidad ha contribuido a mantener el código más limpio y no tiene que construir desde cero las funcionalidades relacionado con la gestión de documento JSON.

1.3.6. Conclusión

En conjunto, todas estas herramientas han permitido abordar el desarrollo de forma ordenada, modular y sostenible. La elección de cada una responde a criterios prácticos, como facilidad de uso, compatibilidad con Unity y eficiencia en entornos de trabajo independientes. Gracias a ellas, ha sido posible mantener una estructura de proyecto clara, adaptar recursos visuales con rapidez y gestionar datos y versiones de manera fiable a lo largo del proceso de desarrollo.

1.4. Estructura de la Memoria

Esta memoria se estructura en siguientes capítulos:

Capítulo 1. Introducción

Se presenta en este capítulo el contexto general del proyecto, la motivación que impulsa su desarrollo y los objetivos principales. También se ofrece una visión general del área técnica abordada y la estructura del documento.

Capítulo 2. Estado de Arte

En este capítulo se presentan los referentes más relevantes relacionados con Magic Overdrive, tanto a nivel de diseño como de tecnología. Se analizan videojuegos similares que han influido en las decisiones de diseño del proyecto, así como las tendencias actuales en el género roguelike y el uso del pixel art como estilo visual. Además, se introducen brevemente las herramientas y tecnologías utilizadas, destacando su presencia y aceptación en el desarrollo de videojuegos independientes.

Capítulo 3. Marco técnico y conceptual

En este capítulo se presentan las bases técnicas y conceptuales sobre las que se construye el proyecto. Se explican las decisiones de diseño que afectan a la estructura general del juego, como el uso de mapas generados dinámicamente, la organización modular de componentes y la aplicación de programación orientada a objetos. También se abordan elementos que conectan lo técnico con lo visual, como la integración de recursos gráficos y el uso de datos externos para configurar distintos aspectos del juego.

Capítulo 4. Implementación Técnica

Aquí se detalla cómo se ha llevado a la práctica el desarrollo del juego. Se describe la estructura del proyecto en Unity, de cómo se implementaron los distintos sistemas: armas, enemigos, mapas, interfaz y gestión de datos. Además, se explican las herramientas utilizadas, la lógica detrás de cada módulo y los problemas técnicos que surgieron durante el proceso, junto con las soluciones aplicadas para resolverlos.

Capítulo 5. Manual de Usuario

Este capítulo se ofrece una guía para el jugador con información sobre controles básicos, interfaz y mecánicas principales, permitiendo al usuario comprender y aprovechar al máximo la experiencia de juego.

Capítulo 6. Conclusiones

Se realiza en este capítulo una síntesis completa de los aprendizajes adquiridos durante el desarrollo del proyecto, tanto a nivel técnico como conceptual.

Capítulo 7. Propuestas de Trabajo Futuro

En este capítulo se exponen posibles mejoras y ampliaciones para futuras versiones del juego, incluyendo nuevos modos, mecánicas, optimizaciones y mejoras técnicas.

Capítulo 8. Bibliografía

Se enumeran todas las fuentes consultadas, tanto para la implementación del proyecto como para la elaboración de esta memoria.

Capítulo 9. Anexo

En esta sección se incluye un enlace al ejecutable del videojuego y al repositorio del proyecto, con el objetivo de permitir su descarga, prueba y revisión.

Capítulo 2. Estado de arte

El desarrollo de Magic Overdrive surge en un contexto en el que los videojuegos independientes han ganado cada vez más visibilidad, gracias a su libertad creativa, su cercanía con el jugador y su capacidad de proponer ideas frescas sin depender de grandes recursos. Juegos sencillos en apariencia, pero con mecánicas bien pensadas, han logrado destacar por encima de producciones mucho más complejas.

Este proyecto nace con esa misma intención: ofrecer una experiencia directa, dinámica y rejugable, utilizando herramientas accesibles y fórmulas de diseño que han demostrado funcionar. Para ello, se han tomado como referencia distintos juegos actuales y elementos comunes dentro del género, tanto a nivel visual como en la forma en que se construye la jugabilidad.

Antes de entrar en las decisiones técnicas tomadas durante el desarrollo, es importante revisar el tipo de juego al que pertenece Magic Overdrive y entender qué lo hace tan atractivo para muchos jugadores.

2.1. Género roguelike

El término roguelike tiene su origen en Rogue (1980), un juego que marcó una base clara: niveles generados aleatoriamente, muerte permanente y progresión basada en la exploración y el riesgo. Aunque en su momento fue una solución a limitaciones técnicas, este tipo de diseño terminó por crear una experiencia intensa, en la que cada decisión importa y cada partida es distinta.

Con el tiempo, el género ha evolucionado y se ha mezclado con otros estilos. Hoy es común ver roguelikes con elementos de acción, plataformas, estrategia o incluso narrativa. Lo que no cambia es la idea central: el jugador aprende jugando, prueba distintas combinaciones, se adapta al azar y busca mejorar poco a poco, sin necesidad de sesiones largas ni estructuras complejas.

Este enfoque ha conectado especialmente bien con el público de juegos independientes. En lugar de depender de gráficos avanzados o campañas extensas, muchos títulos del género se centran en ofrecer partidas cortas pero

intensas, con mecánicas que animan al jugador a volver a intentarlo una y otra vez. Esta es justamente la base que Magic Overdrive toma como punto de partida.

El éxito comercial de muchos roguelikes en plataformas como Steam refuerza esta idea. Títulos como The Binding of Isaac o Hades han conseguido cifras de jugadores muy superiores a la media de juegos independientes, con una comunidad activa incluso años después de su lanzamiento.

Figura 1: Datos de Jugadores simultáneos del juego “The Binding of Isaac” en Steam

Esta combinación de accesibilidad, desafío y variabilidad constante hace del roguelike convierte en una fórmula especialmente efectiva para juegos pequeños o proyectos personales, ya que permite ofrecer una experiencia profunda con un enfoque de producción manejable.

2.2. Estilo visual: pixel art

La estética visual elegida para **Magic Overdrive** es el pixel art, un estilo gráfico caracterizado por emplear píxeles individuales como elementos básicos para crear imágenes y animaciones, evocando directamente la esencia visual de los videojuegos clásicos de 8 y 16 bits.

Aunque el origen del pixel art está estrechamente vinculado a las limitaciones técnicas de las primeras generaciones de consolas y ordenadores personales, Este estilo ha ganado especial popularidad en los últimos años gracias a juegos reconocidos como Celeste, Dead Cells, Hyper Light Drifter o Stardew Valley, entre otros. Estos juegos han dejado claro que el pixel art no es una técnica del pasado. A través de colores bien elegidos, animaciones fluidas y un diseño cuidado, son capaces de crear mundos visuales con personalidad propia y transmitir sensaciones muy potentes, sin necesidad de grandes alardes técnicos.

Además de su estilo reconocible, el pixel art destaca por su accesibilidad. A diferencia de otros enfoques más exigentes en términos artísticos, este permite crear imágenes sólidas sin necesidad de tener una gran formación en dibujo. Se basa en trabajar con bloques simples y bien definidos, lo que facilita producir resultados consistentes incluso para quienes no vienen del mundo del arte. Esto lo convierte en una opción ideal para desarrolladores independientes o proyectos con recursos limitados.

Otro punto clave es la comunidad que se ha formado en torno a este estilo. Existen miles de recursos disponibles en plataformas abiertas, compartidos por artistas de todo el mundo. Gracias a la coherencia visual que caracteriza al pixel art, es posible combinar gráficos de distintas fuentes sin que el resultado final se vea desordenado o incoherente.

A nivel técnico, el pixel art también ofrece ventajas prácticas. Funciona bien en cualquier resolución, se adapta con facilidad a distintos dispositivos y no depende de efectos visuales avanzados. Además, es un estilo que no envejece: mantiene su atractivo con el paso del tiempo y conecta igual de bien con jugadores nuevos y veteranos.

Por todo esto, Magic Overdrive apuesta por el pixel art no solo como una decisión estética, sino como una elección estratégica. Es una forma de ahorrar tiempo, organizar mejor el trabajo y construir un estilo visual claro, funcional y duradero, perfectamente alineado con las características del juego.

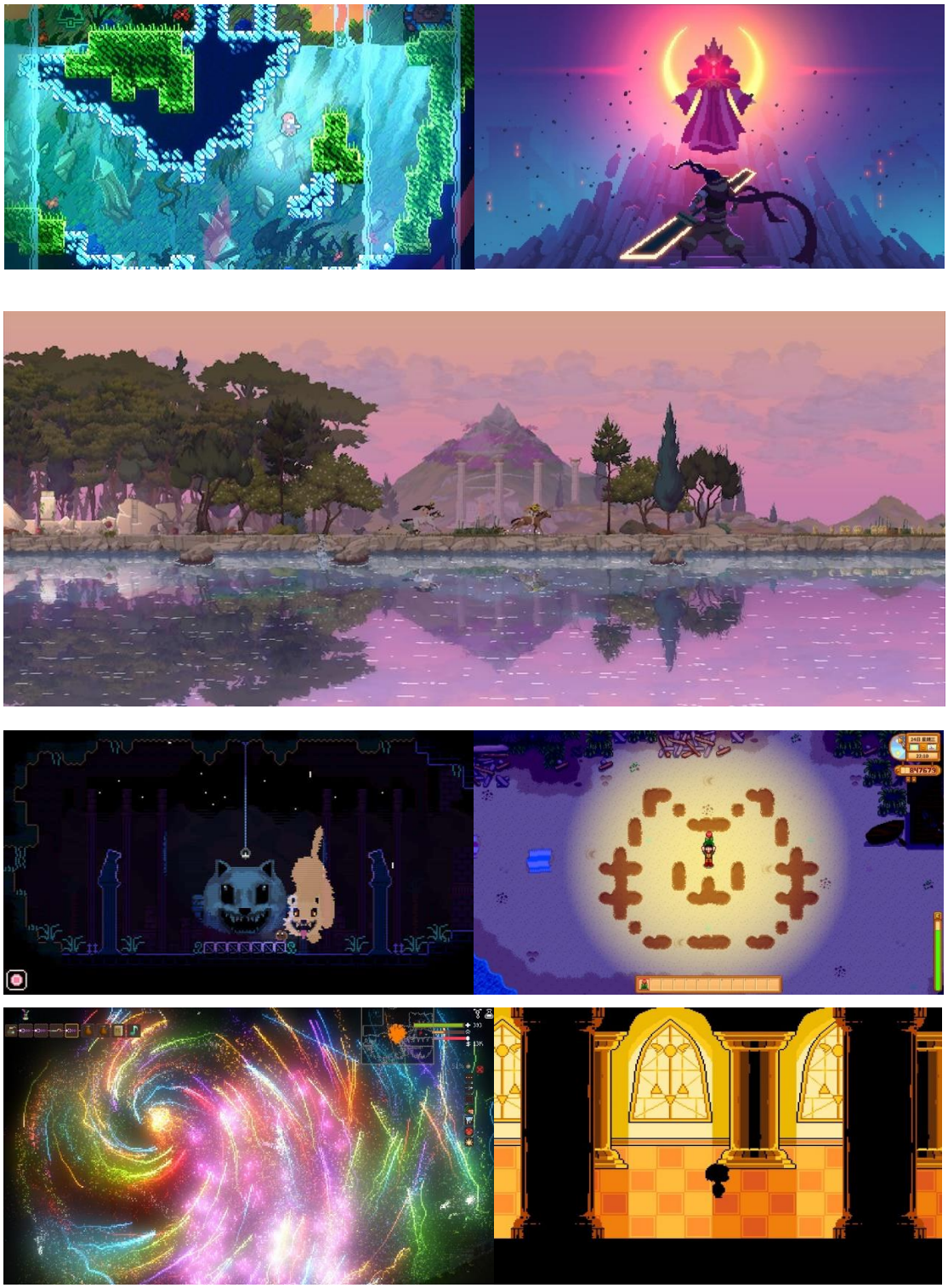

Figura 2: Juego de Pixel Art

2.3. Juegos de referencia

El desarrollo de Magic Overdrive no parte de cero. A lo largo de los últimos años, han surgido muchos juegos que han explorado de forma acertada las mecánicas roguelike, el combate automático o el uso de estilos visuales sencillos pero efectivos. Analizar cómo funcionan estos títulos, qué decisiones de diseño han tomado y por qué han conectado con el público permite entender mejor qué funciona y qué no dentro del género.

En esta sección se recogen algunos de los juegos que han servido como inspiración durante el desarrollo, no solo por sus mecánicas concretas, sino también por su enfoque general, su ritmo, su manera de presentar la progresión o el uso de recursos visuales. Cada uno ha aportado una referencia valiosa a la hora de dar forma a la propuesta de Magic Overdrive.

2.3.1. Vampire Survivors

Vampire Survivors fue uno de los videojuegos que consiguió mayores éxitos independientes en el año 2022, en parte por lo fácil que resulta empezar a jugar. Con una sola mecánica —mover al personaje— y un sistema de combate automático, cualquier persona puede entender y comenzar a disfrutar del juego en pocos segundos, incluso sin experiencia previa en videojuegos.

Sin embargo, detrás de esa entrada simple hay un diseño mucho más elaborado. A medida que avanza la partida, el jugador debe tomar decisiones constantes sobre qué mejoras elegir, qué combinaciones de armas probar o cuándo arriesgarse para conseguir cofres que ofrecen objetos clave. El ritmo de progresión está bien medido, y la curva de dificultad invita a jugar “una más” sin sentir que dos partidas sean iguales.

La variedad de contenido también juega un papel importante: hay múltiples personajes, armas con evoluciones ocultas, condiciones de desbloqueo y mapas con diferentes retos. Todo esto convierte a cada partida (que suele durar entre 20 y 30 minutos) en una experiencia completa, lo suficientemente breve como para no requerir un gran compromiso, pero lo bastante rica como para que el tiempo pase sin que uno se dé cuenta.

Más allá de su éxito comercial, Vampire Survivors demostró que un juego con pocos recursos técnicos puede triunfar si ofrece una mecánica clara, una progresión bien pensada y motivos constantes para volver a jugar.

Finalmente, Otro aspecto clave que refuerza el valor del juego es su compromiso con las actualizaciones. Hoy en día, tres años después de la publicación del juego, Vampire Survivors sigue recibiendo contenido nuevo de forma regular: personajes, armas, fases y modos de juego adicionales. Esta continuidad no solo mantiene activa a su comunidad, sino que también demuestra una visión de desarrollo a largo plazo. Aunque el propio diseño roguelike ya invita a la rejugabilidad, la incorporación constante de novedades aporta frescura y atrae tanto a jugadores nuevos como a quienes ya habían completado gran parte del contenido original.

Figura 3: Vampire Survivors

2.3.2. Brotato

Brotato, un juego que parte de una base muy similar a la de Vampire Survivors, pero introduce cambios importantes que alteran por completo el ritmo y las sensaciones que transmite. Su propuesta es más rápida, más agresiva y enfocada en la toma de decisiones tácticas en poco tiempo. Las partidas son mucho más cortas, lo que hace que la acción se concentre desde el primer minuto y no haya tiempos muertos.

Una de sus principales innovaciones es el sistema de armas: el jugador puede equipar hasta seis al mismo tiempo, combinando efectos, rangos y estilos de ataque. A esto se suma una tienda entre rondas, donde se deben gestionar los recursos con cuidado para decidir qué mejorar, qué conservar y qué adaptar según el tipo de enemigos que irán apareciendo.

El mapa también es más reducido, lo que obliga a un movimiento constante y preciso. Esto aporta un componente de tensión diferente al de otros títulos del género, donde la evasión puede ser más amplia. Aquí, el espacio limitado hace que cada segundo cuente, y que los errores se paguen caro.

A nivel de contenido, Brotato incluye múltiples personajes con estilos y atributos únicos, desbloqueables mediante logros que premian distintos tipos de jugabilidad. Esa variedad no solo alarga la vida del juego, sino que anima a experimentar con combinaciones que ofrecen retos completamente distintos.

Figura 4: Brotato

2.3.3. The Binding of Isaac: Rebirth

The Binding of Isaac: Rebirth, uno de los referentes más popular dentro del género roguelike moderno. Aunque su jugabilidad se basa en mecánicas clásicas de disparos en 2D y exploración de mazmorras, lo que lo distingue es la profundidad de su sistema de objetos, la enorme cantidad de combinaciones posibles y la forma en que cada partida puede desarrollarse de forma impredecible.

En cada recorrido, el jugador se enfrenta a salas generadas de forma procedural, con enemigos, jefes finales y objetos completamente aleatorios. A lo largo de la partida, va adquiriendo mejoras (algunas con efectos positivos, otras con consecuencias inesperadas) que alteran radicalmente la manera en que se juega. Esta imprevisibilidad constante es uno de los grandes atractivos del juego, haciendo que cada partida sea única y completamente impredecible.

Otro punto importante es la forma en que el juego combina dificultad y progresión. A pesar de su alto nivel de exigencia, el sistema de desbloqueos permanentes (personajes, ítems, modos de juego) ofrece una sensación clara de avance. Esto genera una motivación constante por volver a jugar, descubrir nuevas interacciones entre objetos y superar desafíos cada vez más complejos.

A más de una década de su primera versión, The Binding of Isaac sigue siendo actualizado y expandido con nuevas expansiones y ajustes. Su longevidad se muestra que cuando se combinan bien el diseño procedural con un sistema de progresión profundo, el resultado es un juego con capacidad de enganchar más de cientos de horas.

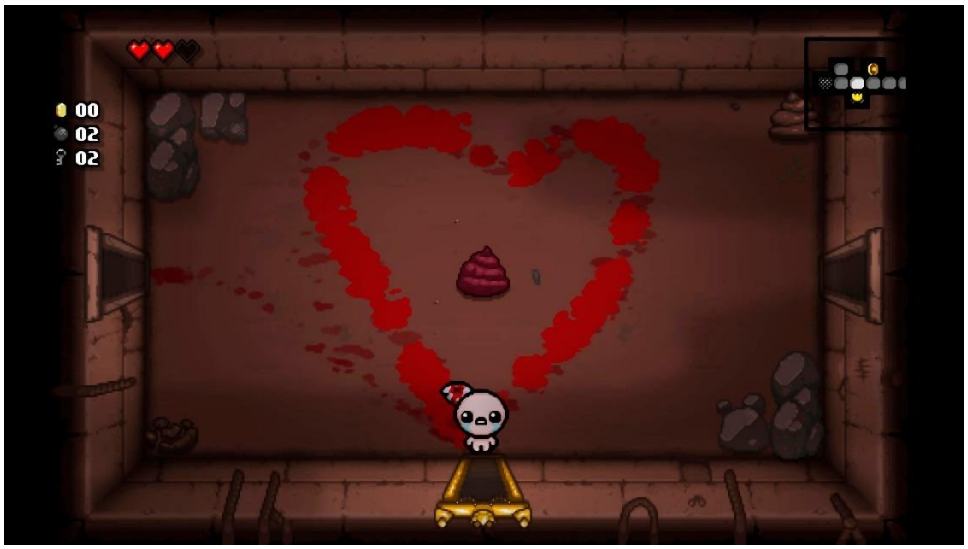

Figura 5: The Binding of Isaac Rebirth

Capítulo 3. Diseño y estructura del juego

3.1. Visión general del juego

Magic Overdrive es un juego de acción roguelike en 2D donde el jugador controla a un personaje que debe sobrevivir a oleadas de enemigos durante una partida de duración limitada. A lo largo del tiempo, el personaje obtiene armas y mejoras que aumentan su capacidad ofensiva y defensiva, mientras el número y la dificultad de los enemigos escalan progresivamente.

El núcleo jugable gira en torno a un sistema de combate automático: el jugador solo se encarga de mover al personaje por el escenario, esquivar enemigos y recoger orbes elementales. El resto —ataques, generación de proyectiles, recolección de experiencia— se realiza de forma automatizada, lo que permite al jugador centrarse en posicionarse y tomar decisiones tácticas sobre qué mejoras escoger y cuándo arriesgarse.

Este tipo de dinámica genera un bucle de juego muy claro: avanzar, sobrevivir, mejorar. Cada partida es distinta gracias a la aleatoriedad en las armas, combinaciones y elementos de progresión. El diseño busca ofrecer una experiencia ágil, pero con suficiente profundidad como para invitar a repetir.

3.2. Estructura de escenas y navegación

El juego está estructurado en dos escenas principales: la escena de selección y la escena de juego. Al iniciar la aplicación, el jugador accede primero a la escena de selección, que actúa como centro de preparación previa a cada partida. En ella se concentran todas las pantallas necesarias para configurar la experiencia de juego: el menú principal, la selección de personaje, la selección de mapa y un apartado de mejoras permanentes (bonus upgrade), donde se pueden gastar monedas obtenidas en partidas anteriores para desbloquear ventajas pasivas.

La navegación entre estos apartados se realiza de forma libre mediante botones, sin pasos forzados ni rutas fijas. Cada sección funciona como un panel independiente que puede activarse o desactivarse según la opción elegida por el jugador. A nivel interno, un sistema de gestión centralizado se encarga de coordinar las transiciones y mantener la coherencia lógica entre los diferentes estados.

Figura 6: Menú principal

Nota: Pantalla de inicio del juego, donde el jugador puede acceder al menú principal, seleccionar personaje, elegir mapa o gestionar mejoras permanentes antes de comenzar la partida.

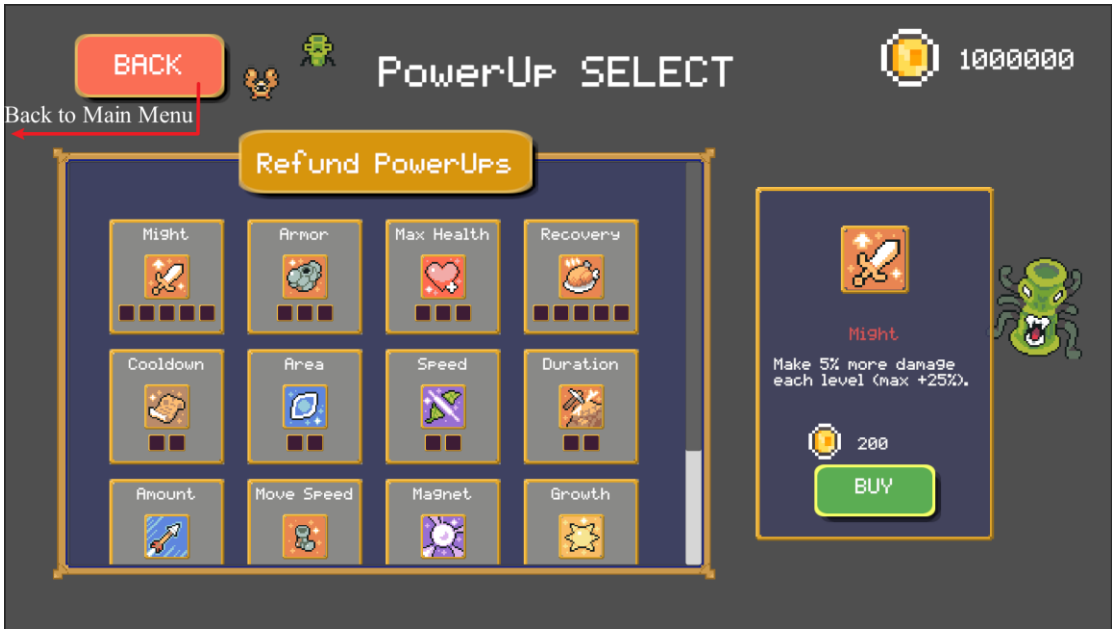

Figura 7: PowerUP Panel

Nota: Panel de mejoras permanentes donde el jugador puede gastar monedas obtenidas en partidas anteriores para desbloquear ventajas pasivas que se aplican al inicio de cada partida.

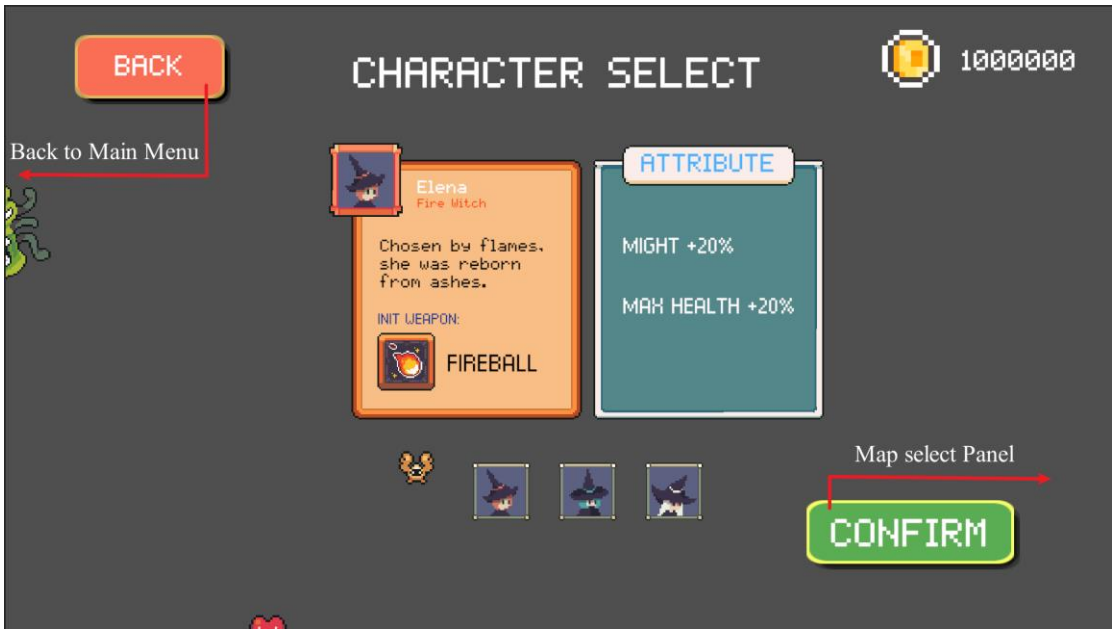

Figura 8: Role Select Panel

Nota: Panel de selección de personaje, donde se muestran los detalles al tener un personaje seleccionado, como el nombre, icono, descripción, arma inicial y atributos únicos asociados.

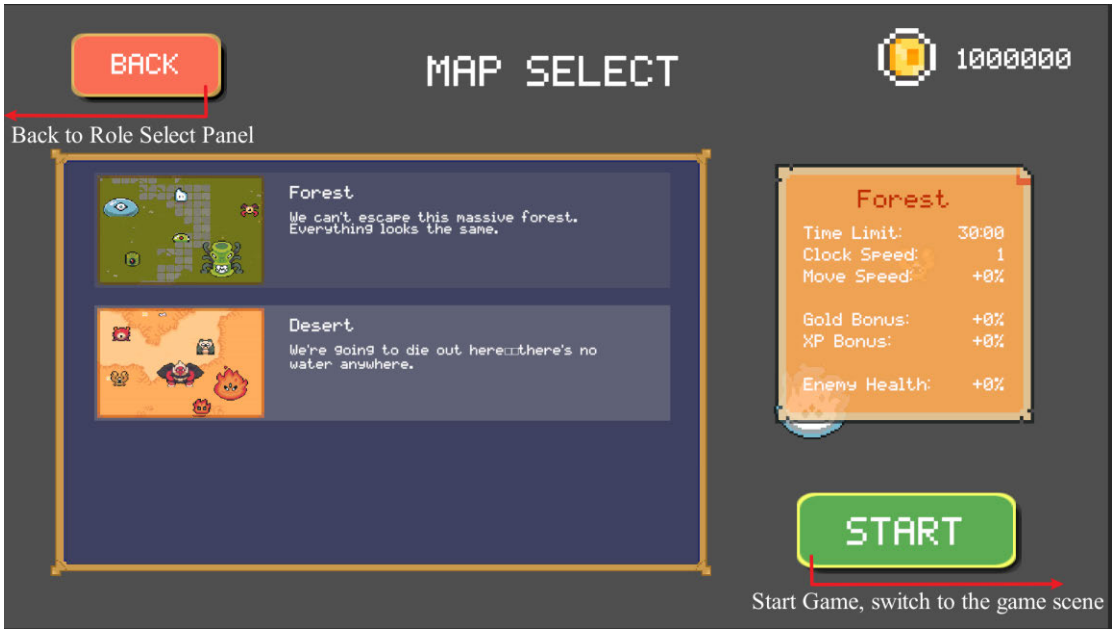

Figura 9: Map Select Panel

Nota: Panel de selección del mapa, donde se muestran los detalles al tener un mapa seleccionado, como el nombre, icono, descripción y las informaciones relevantes.

Una vez realizadas todas las selecciones, el jugador puede iniciar la partida pulsando el botón de start. Esto provoca el cambio a la escena de juego, donde se genera dinámicamente el mapa, se instancia el personaje elegido y se cargan las mejoras permanentes adquiridas.

Durante la partida del juego, el jugador puede pausar el juego en cualquier momento usando la tecla Esc o un botón en pantalla. Desde ahí, puede elegir entre continuar o finalizar la partida. Al terminar, se muestra un resumen con estadísticas clave: número de enemigos derrotados, nivel alcanzado, tiempo de supervivencia y monedas obtenidas. Al confirmar, el juego regresa automáticamente a la escena de selección, permitiendo iniciar una nueva partida o realizar otras acciones.

Figura 10: Pause Panel

Nota: Pantalla de pausa del juego, donde se puede elegir entre continuar o finalizar la partida. También se muestran los objetos obtenidos, junto con su nivel actual y nivel máximo.

Figura 11: GameOver Panel

3.4. Esquema de flujo

Para complementar la explicación estructural del juego, este apartado incluye un esquema visual que representa el flujo principal de una partida completa. A través de este diagrama se resumen las diferentes etapas del juego, desde el menú inicial hasta el fin de partida, incluyendo los procesos intermedios como la selección de personaje, mapa, mejoras y el desarrollo del combate.

Este tipo de representación gráfica permite a entender de forma más clara y rápida la lógica general del funcionamiento del juego, así como las conexiones entre sus distintas escenas y sistemas principales. Servirá tanto para tareas de mantenimiento como para futuras ampliaciones del proyecto.

A continuación, se presenta el diagrama que resume este flujo:

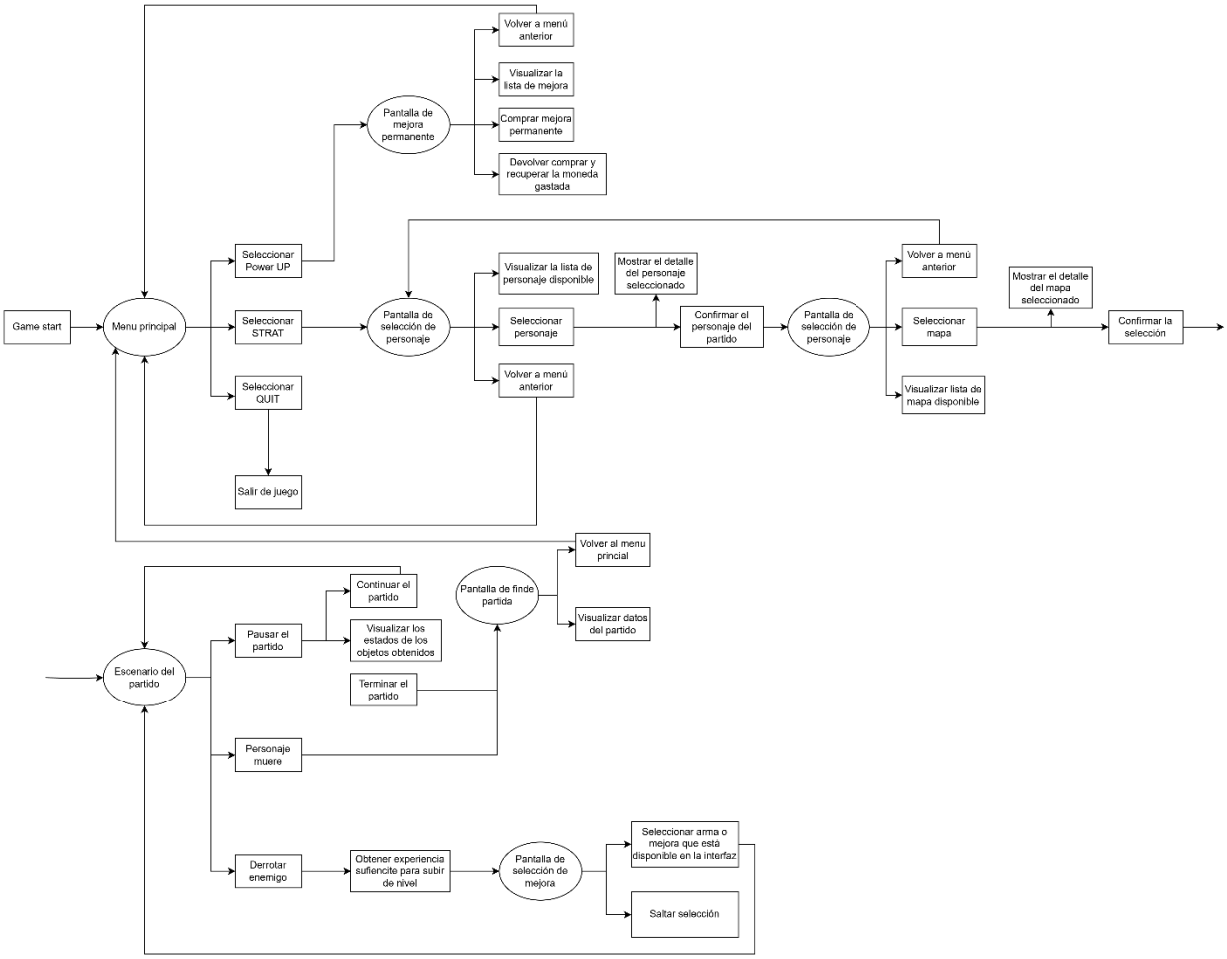

Figura 12: Flujo completo del juego

Capítulo 4. Implementación técnica

Después de definir la estructura general y el diseño conceptual del juego, este capítulo se centra en el desarrollo técnico de sus componentes principales. Aquí se describen con mayor detalle las decisiones de implementación, la lógica detrás de los sistemas clave del juego y la forma en que se ha estructurado el proyecto dentro de Unity.

Cada subapartado aborda un módulo específico, como la generación de enemigos, el sistema de mejoras, la creación del mapa o la gestión de datos, explicando no solo qué hace cada uno, sino también cómo se ha resuelto desde el punto de vista de programación. El objetivo es ofrecer una visión clara del funcionamiento interno del juego y de las soluciones aplicadas a lo largo del desarrollo.

4.1. Sistema de enemigos y oleadas

4.1.1. Creación y gestión de enemigos

El proceso de creación de enemigos en el juego parte de la obtención de sprites y texturas de enemigos 2D, disponibles en bancos de recursos en línea. A partir de estas imágenes se construyen los prefabs correspondientes en Unity, los cuales contienen la configuración básica del enemigo y su comportamiento en escena.

A diferencia de sistemas más complejos basados en animators, el componente de Unity que contiene variedad de funcionalidades en gestión de la animación, en este proyecto se optó por una solución más ligera y controlada para la animación de los enemigos, se resuelve mediante un pequeño script que alterna entre distintos sprites a intervalos definidos, simulando movimiento. Este script también incluye ligeros efectos de escalado, lo que permite dotar a los enemigos de una animación sencilla pero efectiva sin apenas impacto en el rendimiento.

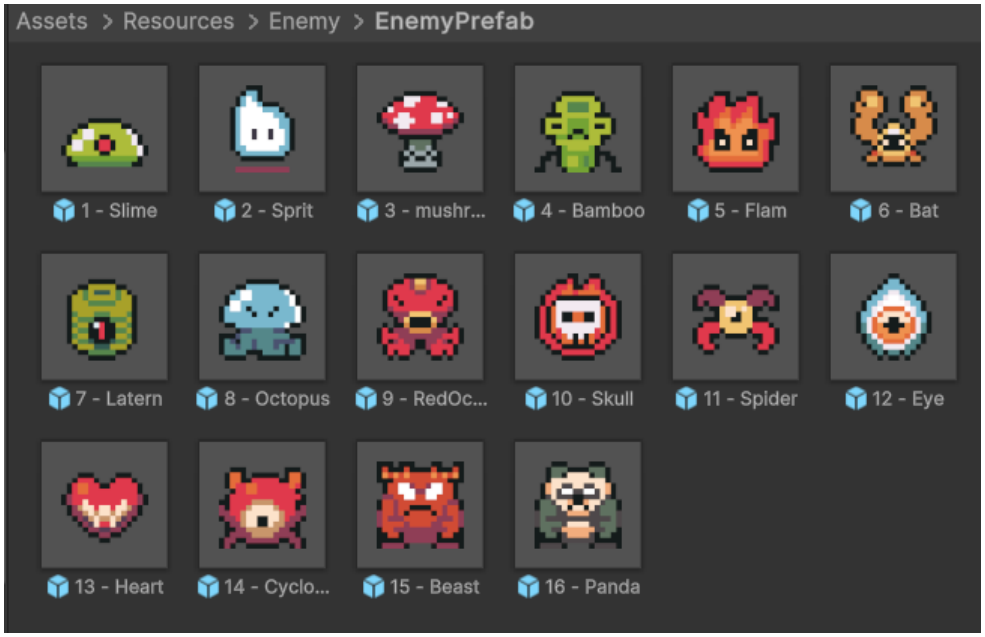

Figura 13: Prefab de los enemigos

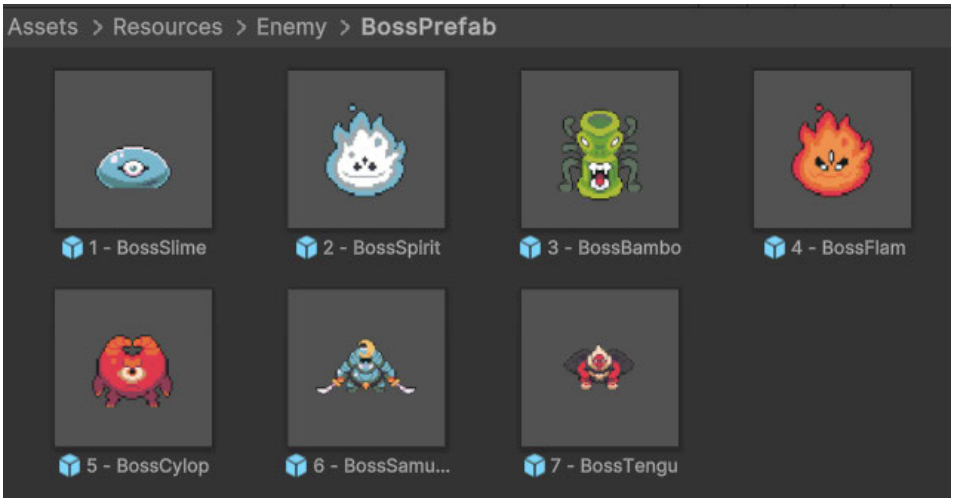

Figura 14: Prefab de los jefes finales

Todos los prefabs generados son organizados en una carpeta específica dentro del proyecto, permitiendo su acceso rápido y una gestión más limpia durante el desarrollo.

A nivel funcional, cada enemigo está compuesto por una serie de parámetros fundamentales: velocidad de movimiento, puntos de vida, daño de ataque, valor de experiencia otorgada y cantidad de monedas que deja al ser derrotado. Estos datos determinan el comportamiento y el valor táctico de cada enemigo durante una partida.

id	name	damage	maxHealth	moveSpeed	knockBackTime	hitWaitTime	experienceDrop	coinDrop	coinDropRate
1	Slime	3	15	1	0.5	1	5	2	0.2
2	Spirit	3	20	1	0.5	1	5	2	0.2
3	mushroom	3	25	1	0.5	1	10	2	0.2
4	Bambo	4	30	1	0.5	1	15	2	0.2
5	Flam	4	35	1	0.5	1	15	2	0.2
6	Bat	5	50	1	0.5	1	20	2	0.2
7	Latern	5	40	1	0.5	1	25	2	0.2
8	Octupus	5	45	1	0.5	1	30	2	0.2
9	RedOctopus	6	60	1	0.5	1	35	2	0.2
10	Skull	8	80	1	0.5	1	35	2	0.2
11	Spider	8	100	1	0.5	1	40	2	0.2
12	Eye	10	120	1	0.5	1	40	2	0.2
13	Heart	12	150	1	0.5	1	50	2	0.2
14	Cyclope	12	200	1	0.5	1	55	2	0.2
15	Beast	12	250	1	0.5	1	60	2	0.2
16	Panda	14	300	1	0.5	1	70	2	0.2

Figura 15: Excel que contiene los parámetros de cada enemigo

Para facilitar la modificación y el mantenimiento de estos valores, se ha optado por centralizar todos los datos en un archivo de Excel. Posteriormente, estos datos se exportan y convierten a formato JSON, lo que permite su lectura directa desde el juego mediante scripts específicos. Esta estrategia agiliza enormemente el ajuste de valores durante la fase de pruebas y balanceo.

En conjunto, esta estructura flexible y controlada permite mantener un equilibrio entre la eficiencia técnica y la facilidad de expansión, lo cual resulta fundamental para un proyecto independiente en constante evolución.

4.1.2. Gestión de oleadas y la generación de enemigos

La generación de enemigos en el juego se basa en un sistema de oleadas progresivas que incrementan tanto en frecuencia como en dificultad a medida que avanza la partida. Para ello, se ha implementado una lógica que toma como referencia un conjunto de datos por oleada: tipo de enemigos, cantidad total y tiempo de aparición entre cada uno. A partir de estos parámetros, se activa un generador que instancia los enemigos fuera del campo de visión del jugador, en posiciones aleatorias alrededor de la pantalla, generando así una sensación constante de presión y movimiento.

La información que define el comportamiento de cada oleada se almacena mediante un ScriptableObject, una estructura de datos que Unity permite serializar de forma independiente al resto de los objetos del juego.

Esta herramienta facilita mantener las configuraciones organizadas, modificarlas sin alterar el código y reutilizarlas fácilmente en diferentes sesiones o modos. El uso de un ScriptableObject para representar las oleadas permite un control más detallado sobre la dificultad y ritmo del juego, además de simplificar la edición desde el editor.

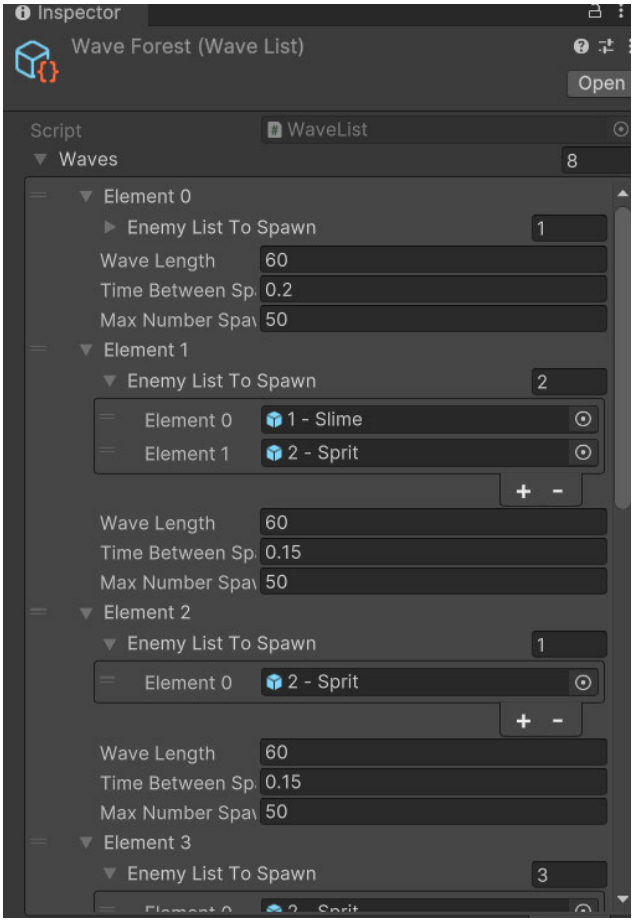

Figura 16: Estructura de ScriptableObject creado para control de oleada

Una vez instanciados, los enemigos están programados para desplazarse automáticamente hacia la posición actual del jugador, lo que mantiene la acción constante y obliga a moverse estratégicamente por el mapa. El sistema asegura que, sin necesidad de una inteligencia artificial compleja, se genere una tensión continua y un flujo de juego fluido.

Figura 17: Forma de Instanciar enemigos

Para optimizar el rendimiento del juego y evitar una acumulación innecesaria de objetos en escena, los enemigos que se alejan demasiado del jugador, específicamente, cuando salen de los límites visibles de la cámara durante cierta distancia, son eliminados automáticamente. Esta mecánica permite mantener bajo control el uso de recursos sin afectar la experiencia de juego, ya que los enemigos activos siempre están en el campo de acción del jugador.

4.2. Sistema de armas y mejoras

4.2.1 Sistema de mejoras

Las mejoras (o bonus) en el juego provienen de tres fuentes distintas, pero comparten una lógica común: modifican estadísticas del personaje o de sus armas, y afectan directamente al rendimiento durante la partida. Estas fuentes son: las mejoras permanentes compradas fuera de partida, los atributos del personaje seleccionado y las mejoras que se obtienen durante el juego al subir de nivel.

A pesar de compartir los mismos tipos de efecto, como velocidad de movimiento, daño, área de ataque, recuperación de vida, entre otros, cada módulo de mejora se gestiona de forma separada. Esta separación permite evitar errores de acumulación y facilita el control de balance, ya que cada componente se calcula de forma independiente antes de aplicar el total al personaje o arma correspondiente.

- **Mejoras permanentes**: adquiridas mediante monedas obtenidas durante las partidas, ofrecen ventajas pasivas que se conservan entre sesiones. El jugador puede comprarlas desde el panel de mejoras fuera del juego.

Figura 18: Mejora permanente

- **Atributos del personaje**: cada personaje tiene un conjunto fijo de valores iniciales, que pueden representar tanto bonificaciones como penalizaciones. Esto contribuye a definir estilos de juego diferentes.

Figura 19: Atributo del personaje

- **Mejoras por nivel**: durante la partida, al subir de nivel, el jugador elige entre varias mejoras aleatorias. Estas afectan a las mismas variables que las anteriores, pero tienen una duración limitada a la partida actual.

Figura 20: Mejora seleccionada al subir de nivel

Al final del proceso, todos los valores de bonificación se combinan sumando los efectos de cada fuente, aplicando el resultado final sobre las estadísticas base. Este enfoque modular mejora el control y permite mantener la flexibilidad para introducir ajustes o nuevas fuentes de mejora en el futuro.

4.2.2. Sistema de armas

Las armas en el juego constituyen el núcleo del sistema de combate. Cada una tiene un comportamiento específico, un patrón de ataque único y una progresión por niveles que afecta a su daño, velocidad, duración, cantidad, tiempo de recarga y alcance. Todas las armas se activan de forma automática, lo que permite al jugador centrarse en el posicionamiento y la estrategia, mientras el sistema se encarga de gestionar los disparos y efectos en segundo plano.

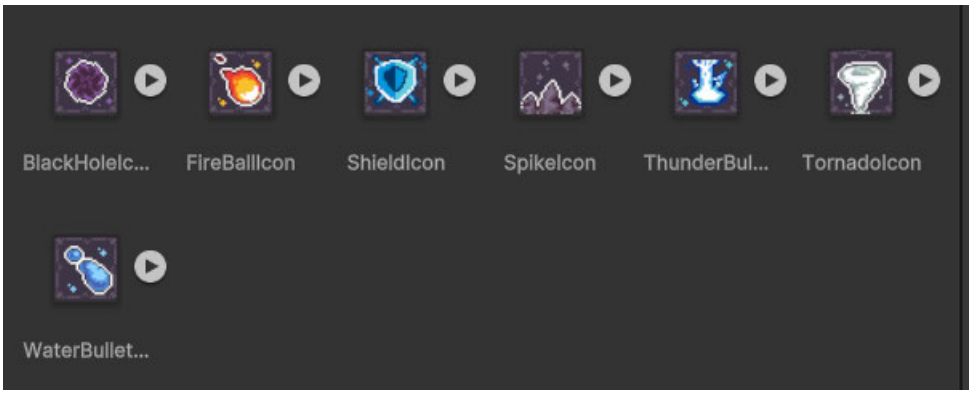

Figura 21: Tipos de Armas diseñada para versión Demo

Al iniciar una partida, el personaje seleccionado comienza con un arma base predeterminada, asociada a su perfil. A medida que avanza la partida y se suben niveles, es posible obtener nuevas armas o mejorar las que ya se tienen, eligiendo entre varias opciones aleatorias. Cada arma puede subir de nivel de forma individual, lo que incrementa sus efectos o modifica su comportamiento de forma progresiva.

El diseño de las armas está pensado para ofrecer variedad en estilo de juego: algunas están orientadas a controlar el espacio, otras priorizan el daño directo, y otras busca realizar daño en área o proteger al jugador. Esta diversidad permite al jugador experimentar con diferentes combinaciones en cada partida, lo que refuerza la rejugabilidad del título.

A continuación, se describen la mecánica de las armas implementadas para la versión Demo que está actualmente el juego, incluyendo su función principal, patrón de ataque y evolución a lo largo de la partida.

1. Fire Ball

Fire Ball es un arma elemental de corto alcance que genera uno o varios proyectiles girando alrededor del jugador. Cada proyectil inflige daño al entrar en contacto con un enemigo, lo que convierte a esta arma en una opción ideal para eliminar enemigos cercanos o protegerse en situaciones de alto riesgo.

Su funcionamiento se basa en un sistema de rotación continua en torno al personaje, creando un anillo ofensivo que actúa como barrera móvil. A medida que el jugador obtiene mejoras, el número de proyectiles activos aumenta (amount), ampliando así la frecuencia y el área de cobertura.

Además de la cantidad, otros parámetros clave como la duración (duration), que determina cuánto tiempo permanecen en pantalla, y el tiempo de recarga (cooldown), permiten ajustar el ritmo y la presencia del arma en el campo, adaptándose a distintos patrones de juego. También puede beneficiarse de mejoras generales como el daño, la velocidad o el área, lo que refuerza su capacidad para mantener alejados a los enemigos y controlar eficazmente el espacio cercano al jugador.

Por el estilo de ataque que está diseñada, se ve especialmente beneficiada por ciertas combinaciones de orbes, lo que permite tener la posibilidad de construir partidas centradas en un estilo defensivo-agresivo basado en el contacto constante con los enemigos.

Figura 22: Fire Ball

2. Water Bullet

Water Bullet es un arma de tipo proyectil que lanza disparos dirigidos automáticamente hacia el enemigo más cercano. Su principal ventaja radica en la capacidad de seguimiento, lo que permite al jugador centrarse únicamente en moverse mientras el arma se encarga de identificar y atacar objetivos sin necesidad de localizar el personaje.

El número de proyectiles lanzados en cada ataque está determinado por el parámetro amount, mientras que el intervalo entre ataques depende del cooldown. Al mejorar estos valores, el arma puede disparar más proyectiles en menos tiempo, aumentando así su capacidad de daño sostenido. Otros atributos como la velocidad de los proyectiles y el daño individual también pueden mejorarse, permitiendo adaptarla a distintos estilos de partida.

A diferencia de armas de área o efecto constante, Water Bullet destaca por su precisión y adaptabilidad, siendo especialmente útil en situaciones donde el jugador necesita moverse constantemente sin perder presión sobre los enemigos.

Figura 23: Water Bullet

3. Tornado

Tornado es un arma que tiene un carácter de ataque mediante lanzar uno o varios torbellinos en direcciones aleatorias. Cada proyectil instanciado se desplaza de forma recta e ininterrumpida durante un tiempo determinado, dañando y empujando hacia atrás a todos los enemigos que encuentra a su paso. Esta característica la convierte en una herramienta ideal para abrir espacio, romper formaciones enemigas o despejar rutas de escape en situaciones de presión.

El número de tornados lanzados en cada ciclo depende del parámetro amount, mientras que el cooldown regula la frecuencia de lanzamiento. Además, los parámetros area y duration influyen directamente en su efectividad: un mayor tamaño permite alcanzar a más enemigos, y una mayor duración amplía la distancia recorrida antes de que desaparezca.

Aunque su trayectoria no es controlada por el jugador, pero su capacidad de alterar el posicionamiento del enemigo aportando un valor especialmente útil, en especial en las combinaciones con otras armas que tiene un ataque defensivo.

Figura 24: Tornado

4. Black Hole

Black Hole es un arma de tipo pasivo que genera una zona circular de daño constante alrededor del jugador. A diferencia de otras armas, no lanza proyectiles ni tiene duración limitada: una vez desbloqueada durante la partida, permanece activa de forma continua hasta el final, infligiendo daño periódico a todos los enemigos que entren en su área de efecto.

El parámetro area define el tamaño del campo de acción, lo que determina el tamaño del radio que puede tener el arma, lo que implica cuántos enemigos pueden ser afectados al mismo tiempo. Por otro lado, damage y cooldown controlan la cantidad de daño infligido en cada pulso y la frecuencia con la que estos se aplican, respectivamente. Al mejorar estos valores, el arma se vuelve capaz de cubrir más espacio del campo visual, realizar más daño a los enemigos y reducir el intervalo de tiempo entre cada ataque.

Debido a las características que posee el arma: el funcionamiento autónomo y persistente, Black Hole es especialmente útil en situaciones con alta concentración de enemigos, ya que castiga con alta frecuencia a cualquier enemigo que se acerque demasiado, reduciendo su vida de forma continua, lo que permite al jugador tener un mejor control sobre el espacio cercano y limitar el movimiento de los enemigos que pretender acercar.

Figura 25: Black Hole

5. Spike

Spike es un arma de área que genera anillos de pinchos alrededor del jugador. Al activarse, se crean varios pinchos fijos en el suelo en una disposición circular, los cuales permanecen en el lugar donde fueron generados y no siguen al personaje. Esto permite dejar zonas de daño persistente sobre el terreno, especialmente útiles para frenar enemigos que persiguen al jugador.

El ataque tiene dos fases diferenciadas: primero, al aparecer, los pinchos infligen un daño inicial a los enemigos en su radio inmediato; luego, durante el tiempo que permanecen activos, siguen haciendo daño a los enemigos que entren en contacto con ellos, aunque con una potencia reducida (alrededor de un tercio del daño inicial). A lo largo de la partida, se pueden generar múltiples círculos de pinchos.

El número de pinchos por cada círculo se define con el parámetro *amount*. El valor de *duration* determina cuánto tiempo permanecen sobre el suelo antes de desaparecer, y el *cooldown* regula la frecuencia con la que se puede lanzar una nueva generación de pinchos.

Dicha arma destaca por su capacidad para causar daño continuo sin requerir reposicionamiento ni puntería por parte del jugador. Resulta especialmente útil en escenarios con muchos enemigos que siguen al personaje, ya que convierte su camino en una trampa constante.

Figura 26: Spike

6. ThunderBolt

ThunderBolt es un arma de ataque a distancia que lanza rayos desde el cielo sobre enemigos seleccionados aleatoriamente en cualquier parte del mapa. Cada rayo impacta al enemigo seleccionado y daña también a los enemigos cercanos dentro de su área de efecto, lo que la convierte en una opción eficaz para mantener una presión constante a lo largo de toda la partida.

Los parámetros principales que definen su comportamiento son amount, que determina cuántos rayos se lanzan en cada activación; cooldown, que regula el tiempo entre cada ataque; y area, que define el alcance de daño de cada impacto. A mayor área, mayor es la probabilidad de afectar a varios enemigos con un solo rayo.

Su principal ventaja es que funciona de manera completamente automática y a gran distancia. Como puede verse en la imagen, los rayos impactan directamente sobre enemigos repartidos por el escenario sin que el jugador tenga que apuntar o cambiar de posición. Esto permite concentrarse en esquivar o reposicionarse mientras el arma sigue haciendo daño en segundo plano.

Sin embargo, ThunderBolt también presenta limitaciones: al no afectar directamente a los enemigos cercanos, no es útil para salir de situaciones de encierro o abrir espacio. Por eso, suele ser más efectiva cuando se combina con armas de corto alcance que puedan cubrir el área inmediata del jugador.

Figura 27: Thunder Bolt

7. Shield

Shield es actualmente la única arma de defensa en el juego, diseñada para proteger al jugador de forma activa una vez activada. Cada vez que el personaje recibe un golpe, se comprueba si dispone de puntos de escudo. Si los hay, se consume uno para anular completamente el daño recibido. Además, se activa una breve ventana de invulnerabilidad y se empuja a los enemigos cercanos sin causarles daño, lo que proporciona un respiro temporal en situaciones de peligro.

El parámetro amount define cuántos puntos de escudo puede acumular el jugador como máximo. Por su parte, cooldown determina el tiempo necesario para recuperar un punto consumido. Visualmente, cada punto de escudo activo se representa mediante un anillo brillante alrededor del personaje, como puede verse en la imagen. Esto permite al jugador saber de un vistazo cuánta protección le queda disponible.

Su valor radica en la capacidad de evitar el daño directo sin necesidad de posicionamiento ni control adicional. Shield puede combinarse con cualquier arma ofensiva, ya sea de corto o largo alcance, y es especialmente útil para facilitar el movimiento entre enemigos o sobrevivir en situaciones de encierro.

Figura 28: Shield

4.2.3. Sistema de armas y mejoras

El sistema de selección de objetos al subir de nivel ha sido una de las partes más complejas del desarrollo del juego, tanto por la lógica que requiere como por la integración de distintos componentes para que funcione correctamente. Su objetivo es ofrecer al jugador una experiencia de progresión coherente, variada y justa, sin perder de vista la aleatoriedad característica del género.

Cada vez que el jugador sube de nivel, el sistema genera una "pool" de objetos posibles. Este pool contiene tanto armas como mejoras pasivas que son elegibles en ese momento, ya sea porque todavía no han sido desbloqueadas o porque pueden seguir mejorándose. A partir de esta lista filtrada, se seleccionan hasta tres opciones de forma aleatoria y se muestran al jugador mediante una interfaz clara, donde puede elegir una.

La lógica para formar este pool tiene en cuenta múltiples factores: si un objeto ya está al máximo nivel, si está bloqueado por condiciones externas, o si el número activado de dicho tipo de objeto ya alcanza a su máximo. La generación y presentación de estas opciones implica coordinar componentes de datos, control visual y lógica de interfaz.

A continuación, se incluye una imagen donde se explica la lógica completa del sistema de generación del pool, con ejemplos de filtrado que se ha aplicado.

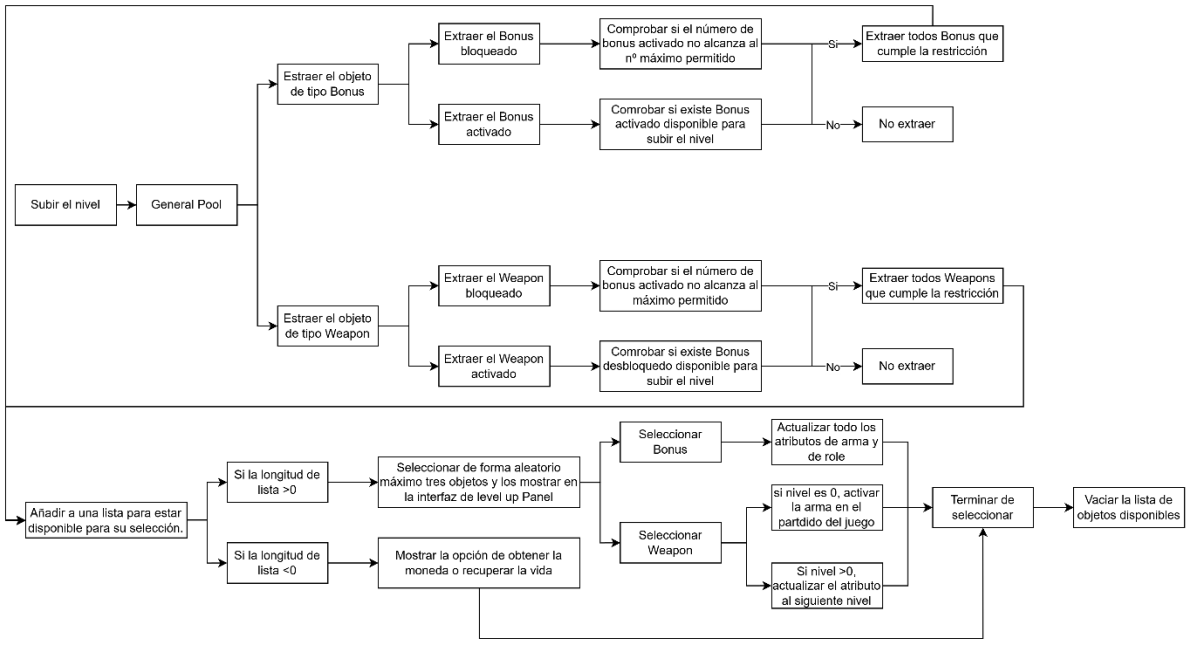

Figura 29: lógica de Object Pool

4.3. Generación del mapa

El sistema de generación del mapa en el juego se basa en una estructura modular que permite crear unos mapas prácticamente infinitos, sin sobrecargar recursos innecesarios. Esta parte del desarrollo combinó herramientas visuales de Unity con un sistema dinámico de control de instancias por código, de ofrecer al jugador una sensación de continuidad y amplitud en el entorno que puede visualizar y sin necesidad de cargar todo el mundo de forma permanente.

El primer paso fue construir los **chunks** o bloques base del mapa que usará como base para instanciar el entorno. Para ello, se utilizaron recursos de tiles en 2D obtenidos en la plataforma como itch.io, que luego fueron organizados manualmente utilizando la herramienta Tilemap de Unity. Cada bloque tiene un tamaño de 32x32 celdas, posteriormente, una vez definido el diseño visual, cada chunk se guarda como un prefab, permitiendo su reutilización durante la generación dinámica.

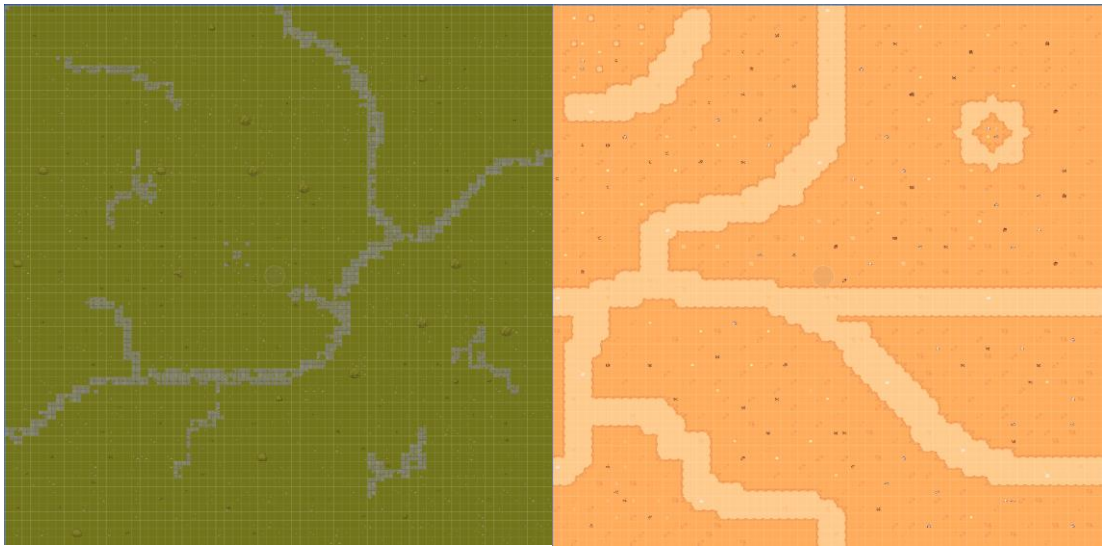

Figura 30: Chunk creado para mapa de bosque y desierto

A partir de estos bloques, se implementó un sistema que permite la generación dinámica del mapa alrededor del jugador en tiempo real. Se revisa continuamente la posición del personaje y, en función de ella, se asegura de que haya una cuadrícula activa de 3x3 chunks centrada en su ubicación. Si el jugador se desplaza lo suficiente, se generan nuevos chunks en las direcciones correspondientes y se eliminan los que han quedado demasiado lejos. Esta lógica

permite mantener la sensación de mundo continuo, sin cargar todo el entorno de una sola vez.

Para gestionar las instancias activas, se utiliza una estructura tipo Dictionary<Vector2Int, GameObject>. Esto permite guardar la posición lógica de cada chunk generado, identificar rápidamente cuáles están activos y decidir si deben mantenerse o eliminarse. Además, esta estructura deja abierta la posibilidad de generar zonas especiales en coordenadas concretas, lo que añade potencial de extensibilidad en el futuro. Aunque por el momento solo se ha implementado un único tipo de chunk como base, el sistema está preparado para incluir variaciones más adelante.

Esta solución no solo permite ahorrar recursos, sino que también facilita el diseño de mapas más ricos y variados sin necesidad de definir todo desde el principio. La combinación de herramientas visuales y lógica modular ha sido clave para que el mapa cumpla su función sin convertirse en un cuello de botella técnico.

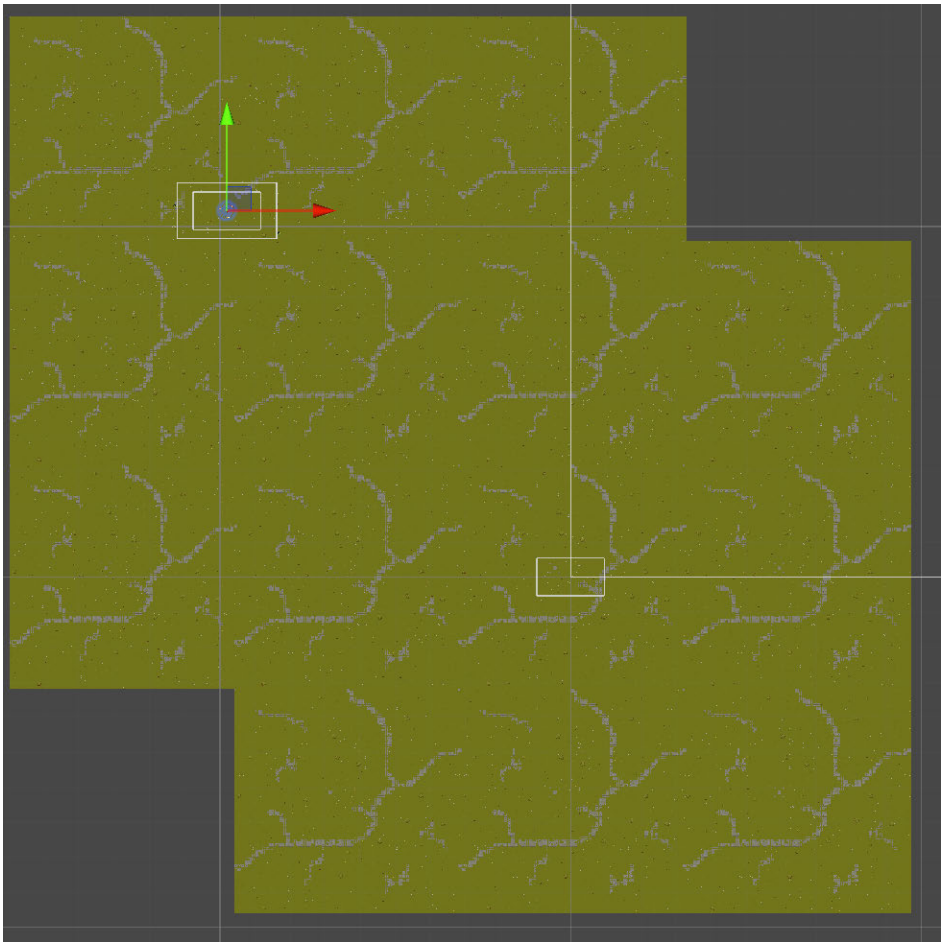

Figura 31: Generación bloque de mapa de forma dinámica

4.4. Interfaz de usuario (UI) y feedback visual

La interfaz del juego ha sido diseñada con el objetivo de ser clara, responsiva y ofrecer al jugador información útil sin distraer de la acción principal. Aunque Unity proporciona herramientas básicas para construir interfaces, fue necesario desarrollar soluciones propias para mejorar la experiencia visual y la usabilidad general.

4.4.1. Comportamiento visual de botones

Unity ofrece componentes estándar como los botones prediseñados (UI Button), pero durante el desarrollo se detectaron algunas limitaciones: las transiciones de estado no siempre eran fluidas y a veces generaban errores visuales.
Para solucionarlo, se implementó un sistema propio de gestión de estados visuales. Este sistema permite cambiar dinámicamente los colores del botón según su estado actual: estado normal, al pasar el cursor por encima (*hover*), al hacer clic (*pressed*) y al soltarlo. Estos cambios visuales ayudan al jugador a percibir que la interfaz está respondiendo correctamente a sus acciones, aportando una sensación más sólida y pulida.

Figura 32: Tres estados de botón

4.4.2. Información contextual en paneles

En varios paneles del juego, como los de selección de personajes o mejora de objetos, se presenta información detallada al hacer clic sobre los distintos iconos o elementos interactivos. Por ejemplo, al seleccionar un personaje se muestran su nombre, atributos y arma inicial; al pulsar sobre una mejora, se despliega una descripción con su efecto.

Este sistema permite mostrar información útil sin cambiar de pantalla ni interrumpir el flujo de la partida. Toda la información aparece de forma inmediata y visualmente integrada, lo que facilita la comparación de opciones y mejora la toma de decisiones por parte del jugador.

Figura 33: Mostración visual de los datos al pulsar el icono del objeto

4.4.3. Feedback visual de daño

Para reforzar la conexión entre las acciones del jugador y sus efectos en el juego, se implementó un sistema visual que muestra el daño infligido a los enemigos mediante números flotantes.

Cada vez que un enemigo recibe daño, aparece un número en la pantalla que representa la cantidad exacta de daño recibido. Estos valores se posicionan justo encima del enemigo afectado y desaparecen tras un breve lapso de tiempo, lo que permite al jugador ver de forma clara e inmediata el impacto de sus ataques sin saturar la pantalla.

Este sistema también ayuda a visualizar con precisión el efecto de las mejoras aplicadas, ya que los números permiten comparar fácilmente el daño antes y después de subir de nivel un arma. Aunque es una función sencilla, añade una capa importante de claridad y satisfacción al momento de jugar.

Figura 34: Números de daño flotando al dañar los enemigos

4.4.4. Sistema de retroceso (knockback)

Con el objetivo de aumentar la sensación de impacto en el combate, se ha añadido un sistema de retroceso que hace que los enemigos se desplacen hacia atrás al recibir un ataque. La distancia del retroceso depende de la fuerza de golpe establecida para cada arma.

Este efecto no solo sirve como elemento visual, sino que también aporta claridad y dinamismo al juego, permitiendo al jugador percibir de forma más directa cuándo un ataque ha conectado. Al mismo tiempo, añade una capa estratégica al diseño de armas: algunas tienen un retroceso fuerte que ayuda a despejar zonas peligrosas, mientras que otras se enfocan más en el daño puro.

Este tipo de respuesta física ante los ataques contribuye a una jugabilidad más inmersiva, haciendo que cada golpe se sienta tangible y diferenciando mejor el estilo de cada arma.

Figura 35: Ejemplo de knockback

4.5. Animaciones del personaje principal

La animación del personaje principal ha requerido un tratamiento especial dentro del desarrollo del juego, diferenciándose claramente de la forma en que se gestionan los enemigos. A diferencia de ellos, que utilizan cambios simples de sprites mediante script para simular movimiento, el personaje jugador necesita transmitir más estados, responder a la interacción del jugador en tiempo real y mantener una coherencia visual que refuerce su protagonismo en pantalla. Por ello, se optó por utilizar el sistema de **Animator** de Unity, que permite definir y gestionar múltiples estados con transiciones claras y parámetros de control definidos.

4.5.1. Motivación y diferencias con los enemigos

El objetivo de gestionar al personaje con Animator no fue solo añadir estética visual, sino al tratarse del único elemento que el jugador controla directamente, su comportamiento debe ser reconocible y reactivo en todo momento.

En el caso de los enemigos, el movimiento es simple y continuo, por lo que bastó con alternar entre los sprites de forma cíclica. Sin embargo, el personaje principal necesita mostrar distintos estados como quieto (idle), corriendo (run), recibiendo daño (get hit) o muriendo (dead), y en todos estos casos, es importante que el cambio de estado ocurra de manera fluida y sin errores visuales.

Además, la animación del personaje sirve también para que el jugador puede tener una **retroalimentación directa**. Por ejemplo, una animación de daño bien ejecutada permite al jugador identificar que ha recibido un impacto, sin necesidad de mirar la barra de vida, lo que contribuye a una experiencia más fluida y conectada con el juego.

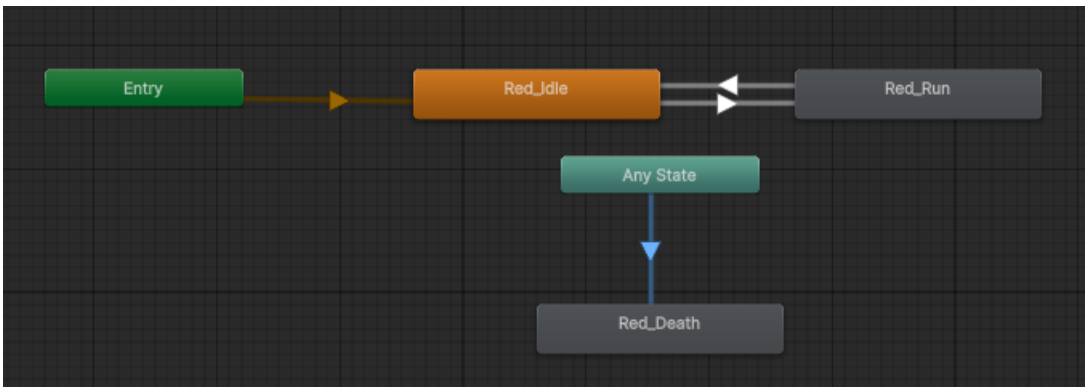

Figura 36: Diagrama de animator

4.5.2. Estados del personaje en Animator

Para estructurar las animaciones del personaje principal, se definieron cuatro estados principales que representan las situaciones más relevantes durante la partida:

- **Idle**: estado por defecto cuando el personaje no se está desplazando. Se representa con una animación sutil o una pose estática.

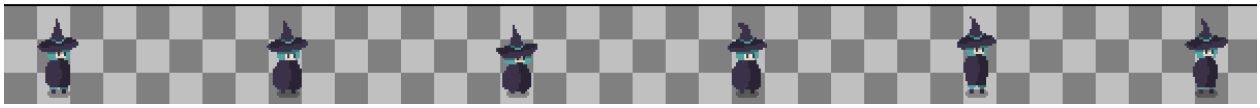

Figura 37: Sprites de estado Idle

- **Run**: activado cuando el jugador pulsa una tecla de dirección y el personaje está en movimiento. Refuerza visualmente la acción del desplazamiento.

Figura 38: Sprites de estado Run

- **GetDamage**: se activa cada vez que el personaje recibe daño. Durante este estado, el sprite del personaje se vuelve semitransparente por un corto periodo de tiempo, lo que indica que ha sido golpeado y, al mismo tiempo, que ha entrado en un estado temporal de invulnerabilidad.

Figura 39: El personaje recibe el ataque

- **Dead**: estado final cuando la vida del personaje llega a cero. Se muestra una animación de desaparición o colapso, marcando claramente el fin de la partida.

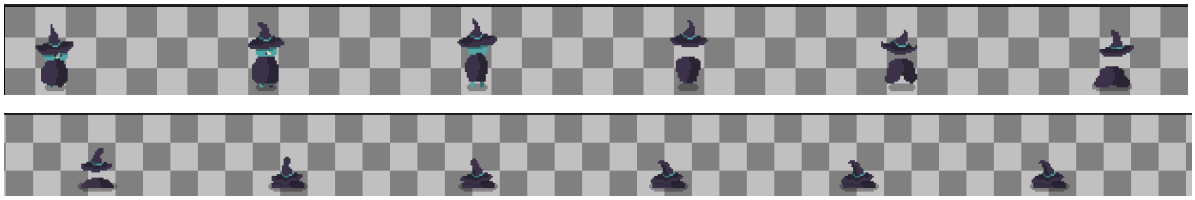

Figura 40: Sprites de estado Dead

Estos estados están conectados entre sí mediante transiciones definidas en el propio **Animator Controller**, en función de variables que se actualizan desde el script del controlador del personaje.

4.5.3. Control y lógica de transición

La transición entre los distintos estados de animación del personaje está gestionada desde el **Animator Controller** de Unity, mediante parámetros definidos que responden a condiciones del juego. Para ellos, se utilizan dos parámetros principales:

- isRun (tipo bool): indica si el personaje se está moviendo.
- isDeath (tipo trigger): se activa al morir el personaje para reproducir su animación final una sola vez.

Tal como muestra la siguiente imagen del sistema Animator, el flujo comienza en el estado Red_Idle. Cuando el jugador comienza a moverse, se activa isRun = true, lo que provoca la transición al estado Red_Run. Al detenerse, la variable vuelve a false, y el personaje regresa a Idle.

Cuando el personaje recibe daño, no se activa una animación en el Animator. En su lugar, se realiza un efecto visual directo en el sprite: el cambio temporal de opacidad, que vuelve al personaje parcialmente transparente por unos instantes. Este efecto comunica de forma clara al jugador que se ha recibido un golpe y que el personaje está en estado de invulnerabilidad temporal.

Finalmente, cuando el personaje muere, se activa el trigger isDeath, que fuerza una transición desde cualquier estado (Any State) al estado Red_Death. Al ser

un trigger en lugar de un bool, se asegura que la animación de muerte no se repita continuamente.

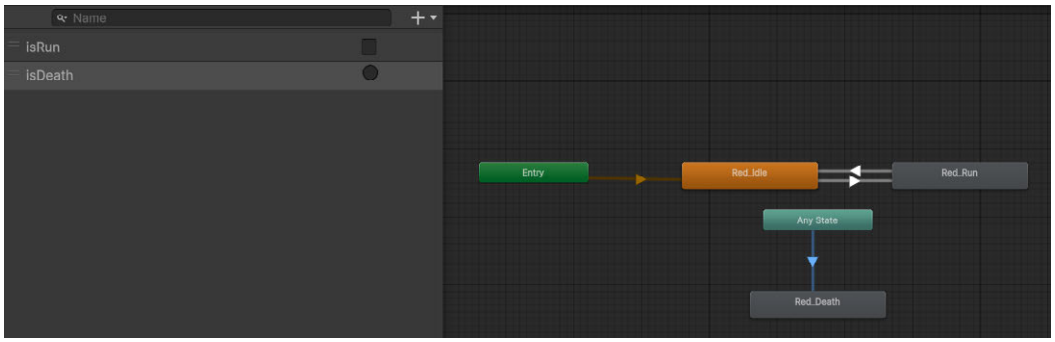

Figura 41: Transiciones entre distintos estados

4.5.4. Integración con el código

La lógica de transición entre estados de animación está integrada en el comportamiento del personaje mediante llamadas desde el código. Para ello se utilizan métodos que actualizan el estado del Animator en función de la situación actual del jugador.

Por ejemplo, en el siguiente método se detecta si el personaje se está moviendo (comprobando si el vector de movimiento es distinto de cero). Si es así, se establece el parámetro isRun como true, lo que activa la animación de carrera; en caso contrario, se vuelve a Idle.

Además, se ajusta también la orientación del sprite según la dirección horizontal del movimiento, escalando el objeto en x.

```
1 个引用 | MinwangLou, 14 天前 | 1 名作者, 1 项更改
private void PlayerMovementAnimation(Vector2 movement)
{
    bool isRunning = movement != Vector2.zero;
    playerAnimator.SetBool("isRun", isRunning);

    if (movement.x > 0)
        playerPrefabInstantiate.transform.localScale = new Vector3(1, 1, 1);
    else if (movement.x < 0)
        playerPrefabInstantiate.transform.localScale = new Vector3(-1, 1, 1);
}
```

Figura 42: Método que activa la animación de correr

Por otra parte, cuando el personaje muere, se ejecuta una corrutina que lanza la animación de muerte. Para ello, se activa el Trigger correspondiente (isDeath), se desactiva el collider del jugador para evitar colisiones y, tras un pequeño

tiempo de espera (para que se vea la animación), se desactiva el objeto del personaje y se muestra el panel de fin de partida:

```
1 个引用|MinwangLou，14 天前|1 名作者，1 项更改
public IEnumerator RoleDeath()
{
    playerAnimator.SetTrigger("isDeath");
    playerDead = true;
    GetComponent<Collider2D>().enabled = false;

    yield return new WaitForSeconds(showDeadAnimationTimer);

    gameObject.SetActive(false);
    SwitchPanelInGame.instance.ShowGameOverPanel();
}
```

Figura 43: Método que gestiona la muerte de personaje

A diferencia de las animaciones de movimiento y muerte, el estado de "recibir daño" no se implementó mediante Animator. En su lugar, se optó por una solución más ligera y directa: la modificación del canal alfa (transparencia) del sprite del personaje.

Cuando el jugador recibe daño, se activa un temporizador durante el cual el sprite del personaje se vuelve parcialmente transparente, indicando visualmente que ha sido alcanzado. Una vez transcurrido el tiempo, la transparente del personaje vuelve a la normalidad.

```
1 个引用|MinwangLou，14 天前|1 名作者，1 项更改
private void TakeDamageAnimation()
{
    if (currentDamageAnimationTimer > 0f)
    {
        currentDamageAnimationTimer -= Time.deltaTime;

        if (!Mathf.Approximately(prefabSprite.color.a, prefabAlpha))
        {
            Color color = prefabSprite.color;
            color.a = Mathf.Clamp01(prefabAlpha);
            prefabSprite.color = color;
        }
    }
    else
    {
        if (!Mathf.Approximately(prefabSprite.color.a, 1f))
        {
            Color color = prefabSprite.color;
            color.a = 1f;
            prefabSprite.color = color;
        }

        currentDamageAnimationTimer = 0f;
    }
}
```

Figura 44: Método utilizado para gestionar la transparenciar del jugador tras recibir daño

4.6. Gestión de datos y serialización

Uno de los pilares técnicos más importantes del juego es el sistema de gestión de datos, está basado casi por completo en archivos JSON. Desde el principio, el proyecto fue pensado para ser fácilmente escalable, y por eso se optó por separar la lógica de juego del contenido mediante estructuras de datos externas. Esto no solo facilita la modificación de parámetros sin tocar el código, sino que permite reutilizar las mismas estructuras en diferentes partes del juego.

El sistema está diseñado para manejar desde información simple, como los valores base de una mejora, hasta estructuras más complejas como listas anidadas de armas o personajes. La mayoría de los datos utilizados en el juego: enemigos, armas, personajes, mejoras, monedas y mapas se almacenan como archivos JSON y se cargan dinámicamente desde el sistema de recursos de Unity.

4.6.1. Conversión desde Excel a JSON

Para facilitar la edición y el mantenimiento de los datos, casi toda la información utilizada en el juego se escribe inicialmente en hojas de Excel. A partir de ahí, se transforma en archivos JSON que el juego puede leer y aplicar en tiempo de ejecución.

En los casos más simples, como los atributos base de enemigos o mejoras, se utiliza una herramienta externa online (tableconvert.com) que permite transformar tablas planas directamente en archivos JSON estructurados. Este método es rápido y suficiente cuando no hay estructuras anidadas.

Sin embargo, para estructuras más complejas como los personajes y las armas, fue necesario desarrollar conversores específicos desde CSV a JSON directamente en Unity, usando scripts a medida.

Por ejemplo, en el archivo que contiene los personajes, cada entrada define no solo la información básica como nombre, descripción, icono o prefab, sino también una sublista de bonificaciones asociadas, que se interpretan como atributos modificados al seleccionar ese personaje. Estas relaciones se mantienen en paralelo dentro del mismo archivo, lo que permite una edición clara sin romper la estructura interna del juego.

	name	avatar	descriptio	slot	record	unlock	unlockCor	initWeapc	rolePrefab_location
1	Elena\n<c	Role/Icon/	Chosen by	6	-1	1	null	1	Prefabs/Role/Red_Witch
2	Lia\n<col	Role/Icon/	Born of a	6	-1	1	null	2	Prefabs/Role/Blue_Witch
3	Freya\n<c	Role/Icon/	Raised in t	6	-1	0	do someth	3	Prefabs/Role/White_Witch

ole	id	name	idBonus	value
1	1	Might	1	0.2
1	2	Max Healt	3	0.2
2	1	Area	6	0.1
2	2	Speed	7	0.1
3	1	Armor	2	1
3	2	Recovery	4	0.2

Figura 45: Estructura de datos de todos los roles

De forma similar, los archivos de armas incluyen no solo sus propiedades generales (nombre, icono, rareza, prefab, etc.), sino también una lista de niveles de mejora, y dentro de cada nivel, otra lista con los atributos que se ven afectados. Además, cada nivel de mejora lleva asociado un texto descriptivo que se utiliza en la interfaz de panel de subida de nivel cuando el jugador mejora esa arma. Esta estructura, lista de niveles más lista de atributos por nivel, sería complicado de mantener sin automatización, y aquí es donde el conversor por código ha resultado esencial.

id	name	icon	rarityWeig	currentLev	maxLevel	prefab	unlock	knockBackForce
1	Fireball	Weapon/I	100	0	5	prefabs/W	1	1

idWeapon	id	upgrateText
1	1	Unlock FireBall
1	2	level 2
1	3	level 3
1	4	level 4
1	5	level 5

idLevel	idAttribute	name	value
1	1	Might	5
1	5	Cooldown	3
1	6	Area	2
1	7	Speed	200
1	8	Duration	3
1	9	Amount	3
2	1	Might	7
2	5	Cooldown	3
2	6	Area	2.5
2	7	Speed	200
2	8	Duration	4
2	9	Amount	3
3	1	Might	7
3	5	Cooldown	2.5

Figura 46: Estructura de datos de un arma

4.6.2. Guardado de progreso y carga de datos persistentes

El juego tiene diseñado e implementado con un sistema de persistencia que sirve para almacenar la información relevante entre sesiones. El objetivo es asegurar que, al volver a abrir el juego, el jugador conserve sus progresos clave: monedas acumuladas, estados de las mejoras permanentes y estado de desbloqueo de los personajes y mapas.

Toda esta información se gestiona de forma centralizada y se guarda como archivos JSON en un directorio de persistencia local. Al iniciar el juego, el sistema comprueba si existen esos archivos; si no los encuentra, genera versiones nuevas con los valores por defecto necesarios para que el juego arranque sin errores.

Los datos que se almacenan actualmente son los siguientes:

- Monedas: guarda la cantidad total de oro conseguido por el jugador, necesario para comprar mejoras fuera de partida.
- Mejoras permanentes: almacena el nivel de cada mejora pasiva y su estado actual.
- Personajes: se registra qué personajes están desbloqueados, con la vista puesta en futuros contenidos que requerirán comprarlos o desbloquearlos mediante logros.
- Mapas: igual que con los personajes, se guarda el acceso a escenarios específicos, aunque por ahora solo se utilice un mapa base.

Este sistema está pensado para ampliarse fácilmente. Añadir un nuevo tipo de dato persistente solo requiere incorporarlo a la estructura del JSON y al sistema de lectura/escritura ya existente. Todo se mantiene dentro de un sistema común, lo que evita dispersión y errores al acceder desde diferentes partes del juego.

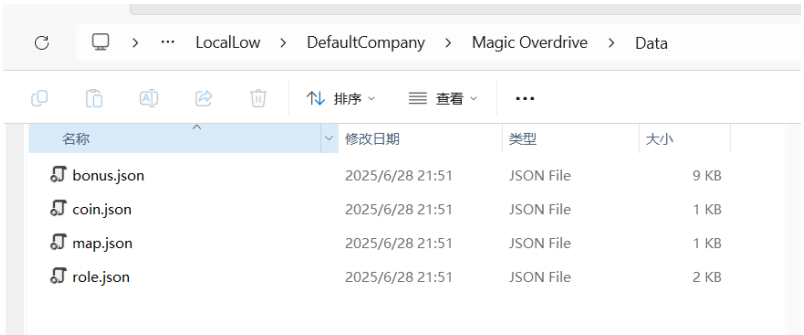

Figura 47: Datos en formato Json guardado en el directorio persistente

4.6.3. Lectura e integración de datos en Unity

En el juego, la mayoría de los archivos de datos se cargan al inicio del juego. Esto incluye la información sobre armas, personajes, mejoras, monedas, mapas y configuraciones generales. Estos datos se almacenan en memoria en estructuras correspondientes, preparadas para ser utilizadas en cualquier momento. Solo unos pocos casos, como los atributos de los enemigos, se cargan más adelante, como al entrar en la escena de juego.

Para facilitar el proceso de lectura y conversión, se utiliza la librería Newtonsoft.Json, que permite convertir directamente un archivo JSON en una instancia de una clase con una sola línea de código. Esta herramienta ha simplificado enormemente el tratamiento de datos, especialmente en estructuras con múltiples niveles o listas anidadas.

Sin embargo, uno de los aspectos más importantes al usar Newtonsoft.Json es mantener una correspondencia exacta entre los nombres y el orden de las propiedades del archivo JSON y las del modelo de clase en C#. Si los campos no coinciden correctamente, la conversión puede fallar o devolver datos incorrectos. Por eso, se ha tenido especial cuidado en asegurar que la estructura de los archivos esté alineada con el código.

Esta combinación de carga anticipada y uso de herramientas externas ha permitido mantener el sistema de datos del juego ordenado, eficiente y preparado para ampliárselo en las versiones futuras, sin tener la necesidad de rehacer código ya implementado.

4.7. Estructura general del proyecto en Unity

La organización interna del proyecto en Unity ha sido pensada desde el principio para facilitar el mantenimiento, la escalabilidad y la lectura del código y los recursos situadas en la carpeta de Resources. A medida que el juego fue creciendo en complejidad, se hizo evidente la necesidad de mantener una estructura clara tanto en la jerarquía de carpetas como en la lógica de scripts y prefabs.

La estructura del proyecto se divide en carpetas principales según su función: scripts, resources, prefabs, escenas, texture, entre otros. Cada carpeta contiene a su vez subcarpetas organizadas por categorías específicas (por ejemplo, los scripts están separados por tipo: Enum, scripts para el escenario principal, scirpts para el escenario del partido, manager, model y utils). Esta división permite localizar y modificar cualquier componente sin necesidad de perder tiempo navegando por archivos mezclados.

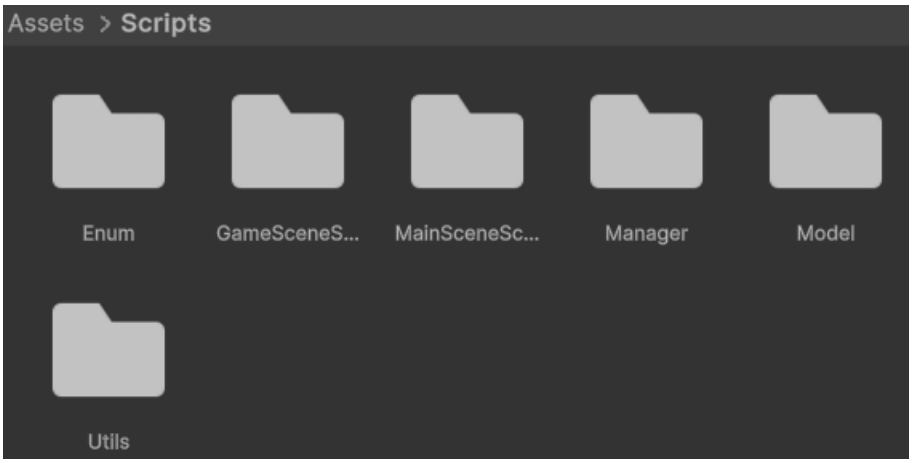

Figura 48: Carpeta Script organizada según la funcionalidad que le corresponde.

En cuanto al código, se ha seguido un enfoque modular: cada sistema (por ejemplo, el sistema de oleadas, el pool de mejoras, o la lógica de selección de personaje) está encapsulado en sus propias clases. Se evita el uso de controladores únicos que gestionen demasiadas cosas a la vez, y se prioriza la separación de responsabilidades. Esto permite mantener el código más limpio, facilita las pruebas individuales y hace más sencillo implementar cambios sin romper funcionalidades ya existentes.

Figura 49: Controladores estructuradas por su propia responsabilidad

Del lado visual, se hace uso intensivo de prefabs para instanciar enemigos, armas, partículas, paneles de interfaz, etc. Todo lo que se repite o necesita parámetros configurables está convertido en prefab, lo que reduce errores y acelera el trabajo en el editor.

En conjunto, esta estructura no solo ayuda a mantener el proyecto ordenado, sino que también sienta las bases para poder seguir ampliándolo sin que se vuelva inmanejable.

4.8. Conclusión del capítulo

El desarrollo técnico del juego Magic Overdrive ha requerido la implementación de múltiples sistemas que interactúan entre sí: desde la gestión dinámica de enemigos y armas, hasta la generación de mapas, la lectura de datos externos y la interfaz de usuario. Cada uno de estos módulos ha sido construido con un enfoque modular y reutilizable, buscando siempre mantener el código claro y fácil de mantener.

Durante el proceso, se tomaron decisiones orientadas a la flexibilidad a largo plazo, como el uso generalizado de archivos JSON, la carga dinámica de datos o la separación lógica de responsabilidades. Muchos de estos sistemas han supuesto un reto, no solo por su complejidad, sino también por la necesidad de integrarlos de forma estable en un proyecto en constante crecimiento.

El resultado es una base técnica sólida que permite seguir ampliando el juego sin necesidad de rehacer estructuras ya existentes. Esta implementación modular es lo que hace posible incorporar nuevas armas, personajes, enemigos o escenarios con un esfuerzo razonable y sin comprometer el equilibrio general del proyecto.

Capítulo 5. Manual de usuario

En este capítulo se presentan los resultados obtenidos tras el desarrollo completo de Magic Overdrive. Más allá del código, los sistemas o las estructuras internas, este apartado tiene como objetivo mostrar de forma visual y concreta cómo se ve y se juega el producto final.

Se muestra en este capítulo diferentes partes del juego con capturas y explicaciones breves que reflejan el funcionamiento real del juego, desde los menús iniciales hasta el combate, pasando por la selección de personajes, mejoras, progresión y UI.

Más que hablar de lo que debería hacer, esta sección trata de enseñar lo que ya está hecho.

Figura 50: Pantalla de inicio

Al entrar al juego, este es la pantalla principal que muestra delante del jugador, les ofrece acceso directo a las secciones clave del juego. Desde aquí se puede iniciar una partida (START), abrir el panel de mejoras permanentes (POWER UP) o salir del juego (QUIT). En la esquina superior derecha se muestra la cantidad total de monedas acumuladas en cada partida, que se puede utilizar para

realizar compra en la tienda de mejora permanente e en futuro servirá también para desbloquear el personaje o mapa.

El botón Delete Save que sitúa en la parte superior izquierda permite al jugador eliminar todos los datos guardados, incluyendo mejoras adquiridas, personajes desbloqueados y progreso general. Cada botón tiene retroalimentación visual en los diferentes estados (normal, al pasar el cursor y al pulsar), manteniendo consistencia en la interfaz.

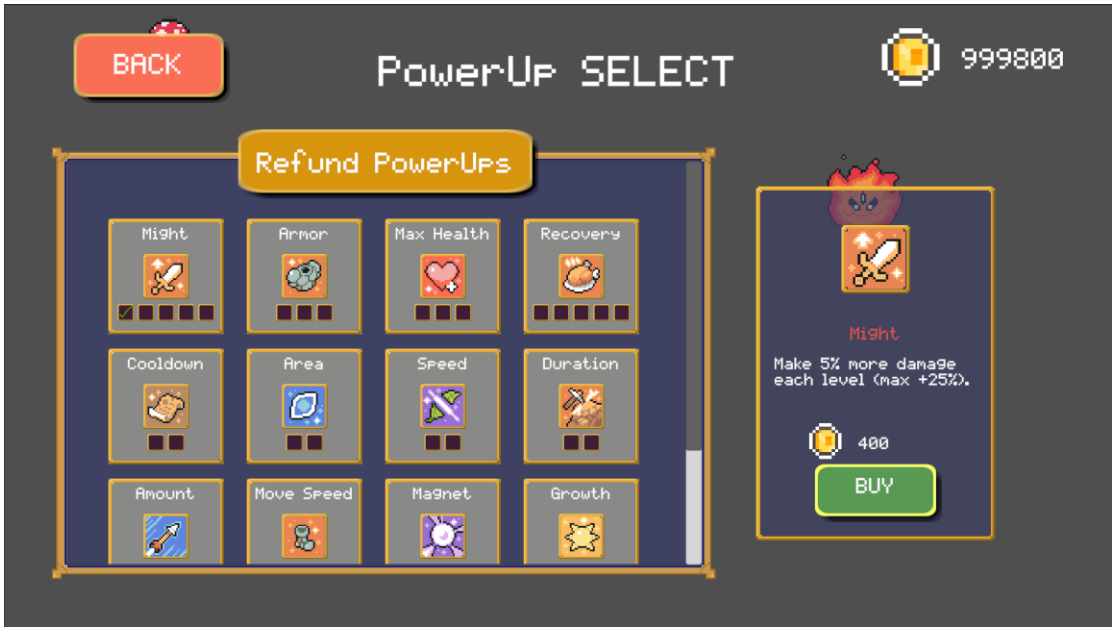

Figura 51: Pantalla de mejoras permanentes (Power Up Panel)

Este panel permite al jugador invertir las monedas obtenidas en partida para adquirir mejoras permanentes. Cada mejora tiene varios niveles y su efecto se aplica automáticamente al comienzo de cada nueva partida.

Las mejoras disponibles se muestran en una cuadrícula, cada una con su icono, nombre y progreso actual representado mediante cuadros de color. Al seleccionar una, aparece a la derecha una descripción detallada del efecto y su coste en monedas. Si el jugador tiene suficiente oro, puede pulsar el botón BUY para subir de nivel la mejora.

El botón Refund PowerUps en la parte superior permite devolver todas las mejoras y recuperar el total de monedas gastadas, útil para probar otras combinaciones o reiniciar la progresión.

Figura 52: Pantalla de selección de personaje

Antes de comenzar la partida, el jugador debe elegir un personaje. Cada uno tiene un arma inicial predefinida y una serie de atributos únicos que afectan directamente a su rendimiento en juego.

En la parte inferior se encuentran los retratos de los personajes disponibles. Al seleccionar uno, se actualiza la información mostrada: a la izquierda aparece su nombre, una breve descripción y el arma con la que empieza; a la derecha se detallan los modificadores de atributos que se aplican desde el primer momento, como aumento de daño, salud máxima o velocidad.

El botón CONFIRM valida la selección y permite pasar al siguiente panel.

Figura 53: Pantalla de selección de mapa

Una vez tener seleccionado el personaje, el juego pasa a la pantalla de elección de mapa. Aquí el jugador puede escoger el escenario que prefiera en el que se desarrollará la partida. Cada mapa tiene sus propias características, tanto visuales como el tipo de enemigos que se instancia.

En el panel de parte izquierda se muestran las opciones disponibles, cada una acompañada de una imagen representativa y una breve descripción correspondiente. Al seleccionar un mapa, se actualiza la información a la derecha, donde se indican parámetros concretos como:

- Límite de tiempo
- Velocidad del reloj del juego
- Modificadores de velocidad de movimiento
- Bonus de oro y experiencia
- Modificador de vida para los enemigos

Estas propiedades pueden hacer que una partida sea más fácil, más exigente o simplemente diferente. Una vez confirmado el mapa que quería jugar, el botón START implica iniciar la partida con la configuración actual seleccionada.

Figura 54: Pantalla de juego

Durante la partida, el jugador controla al personaje en una vista cenital, moviéndose libremente por el mapa mientras aparecen oleadas de enemigos. El combate es automático: las armas se activan y atacan por sí solas según sus propios intervalos, permitiendo al jugador centrarse en el posicionamiento y en la recolección de recursos.

En la parte superior se muestra el nivel actual del jugador, el tiempo transcurrido desde el inicio y la barra de experiencia. A la derecha se visualizan las monedas acumuladas y el botón de pausa, que permite acceder al menú de control. A la izquierda se encuentra el inventario de armas y mejoras obtenidas durante esa partida, que se irá llenando conforme avance el juego.

El diseño está pensado para ser claro y funcional, dejando el centro de la pantalla lo más libre posible para facilitar la lectura del combate y el movimiento entre enemigos.

Figura 55: Pantalla de selección de mejora

Cada vez que el jugador sube de nivel, se interrumpe temporalmente la partida y se muestra este panel de selección. En él se presentan hasta tres opciones de mejora, elegidas aleatoriamente a partir del conjunto de objetos y atributos disponibles en esa partida.

Cada carta muestra el nombre del atributo, su nivel actual y una descripción corta de su efecto. Al seleccionar una opción, se aplica de inmediato y se retoma la partida sin más transición. Estas mejoras incluyen tanto atributos generales (como velocidad, área, cantidad de proyectiles y entre otros) como nuevas armas o evoluciones de las ya obtenidas.

El botón SKIP LEVEL UP permite omitir la mejora en caso de no querer seleccionar ninguna, aunque no suele ser recomendable. Este sistema de progresión incremental aporta variedad y permite construir estrategias diferentes en cada partida.

Figura 56: Menú de pausa

Durante la partida, el jugador puede pausar el juego en cualquier momento presionando el botón en la esquina superior derecha o la tecla ESC. Esto detiene completamente el juego y muestra un menú simple con dos opciones principales: RESUME, para continuar la partida desde donde se dejó, y QUIT, para salir de la sesión actual y volver al menú principal.

A la izquierda permanece visible el panel de estado de los objetos, donde se listan todas las armas y mejoras obtenidas hasta el momento. Cada una muestra su nivel actual y su nivel máximo, lo que permite al jugador visualizar esta información con el juego pausado. Esto facilita tomar decisiones precisas sobre futuras selecciones de mejora o evolución de armas, sin perder el hilo de la progresión.

Este menú cumple una función básica pero esencial: permite al jugador controlar el ritmo de la partida sin interferir con la interfaz general. Además, al pausar el juego, ofrece un momento de respiro durante el partido, permitiendo retomar la acción cuando se sienta preparado para continuar.

Figura 57: Pantalla de fin de partida

Cuando el personaje muere o el jugador decide salir desde el menú de pausa, se muestra la pantalla de fin de partida. En ella se resumen los datos más relevantes de la sesión:

- Total enemigo derrotado
- Jefes finales derrotados
- Tiempo total de supervivencia
- Nivel máximo alcanzado
- Monedas obtenidas durante esa partida

Esta información permite al jugador visualizar el resultado obtenido durante todo el partido y entender su progreso dentro del juego. Al pulsar el botón DONE, se cierra el panel y se regresa a la pantalla de selección inicial, conservando tanto las monedas obtenidas como el estado general de desbloqueo, y dejando todo listo para comenzar una nueva partida según la preferencia del jugador.

Capítulo 6. Conclusión

Desarrollar Magic Overdrive ha sido un proceso largo y exigente, pero también una experiencia enriquecedora en muchos sentidos. Desde la planificación inicial hasta la implementación final, el proyecto me ha permitido aplicar de forma práctica conocimientos técnicos adquiridos durante la carrera, y al mismo tiempo enfrentar desafíos reales que no siempre aparecen en ejercicios académicos.

Uno de los mayores logros ha sido completar un videojuego funcional, con sistemas interconectados como el manejo de armas, enemigos, generación de mapas, gestión de datos en JSON, guardado persistente y una interfaz de usuario clara. Cada uno de estos sistemas no solo requería un desarrollo en la infraestructura del juego, sino también decisiones de diseño que tuvieran en cuenta la experiencia del jugador.

Durante el desarrollo me encontré con problemas complejos: desde cómo organizar el código para que fuera fácil de ampliar, hasta cómo integrar distintos módulos sin que aparecieran errores inesperados. Para resolverlos, tuve que volver a repasar ideas y conceptos que ya conocía. En muchos casos, también fue necesario romper y rehacer partes de la planificación inicial, reorganizar las ideas y finalmente poder alcanzar a una solución que, aunque no es perfecto, pero cumpliera con lo necesario. Durante este camino, Hizo falta mucha paciencia para probar, corregir errores y repetir varias veces hasta que todo funcionara como desea.

Además, trabajar con herramientas como Unity, Visual Studio, Aseprite y GitHub me ayudó a entender mejor cómo prepararme para un proyecto completo y complejo, donde no solo importa que el código funcione, sino también que el diseño encaje bien con la jugabilidad y todo tenga sentido en conjunto para que el jugador puede encontrar la diversión del juego cuando juega. También el manejo de datos externos con Excel y su conversión automatizada a JSON me dio una visión más clara sobre cómo debería estructurar bien la información para que sea fácil de mantener y que sea reutilizable.

A nivel personal, este proyecto me ha enseñado a valorar el proceso creativo más allá del resultado: a veces las ideas no salen como uno espera, pero esa frustración también forma parte del camino. He entendido que desarrollar un videojuego no es solo programar, sino también tomar decisiones, establecer

prioridades entre tareas y saber avanzar incluso cuando no todo está ideal o completo.

Ser sincero de decir, Magic Overdrive no es un juego perfecto, pero representa fielmente el esfuerzo, el aprendizaje y la evolución que fui aprendiendo poco a poco mientras avanzaba con este proyecto. Más allá del resultado final, me quedo con la satisfacción de haber construido algo desde cero, de principio a fin, y con la confianza de poder afrontar futuros retos con una base más sólida y una mentalidad más abierta.

Capítulo 7. Propuestas de Trabajo Futuro

Aunque ahora el juego ya tiene todo lo necesario para funcionar bien y ofrecer partidas completas, todavía hay muchas ideas que me gustaría seguir desarrollando en el futuro para hacerlo más completo y entretenido.

Una de las cosas que tiene mayor prioridad sería añadir más contenido jugable: personajes con habilidades distintas, armas nuevas con mecánicas diferentes, y mapas con efectos o condiciones que cambien el ritmo del partido y la forma de jugar. También estaría bien crear más tipos de enemigos y jefes finales con comportamientos únicos, para que cada partida tenga más variedad y que sea más desafiante.

En cuanto a mecánicas, todavía tengo pendiente el sistema de evolución de armas, una de función fundamental para este tipo de juego, que permitiría mejorar ciertas armas si se cumplen condiciones concretas. Además, me gustaría incluir objetos especiales que solo se consigan al derrotar a ciertos enemigos o romper elementos destruibles del escenario.

Otro punto importante sería meter algún tipo de sistema de logros o misiones, que sirvan como objetivos secundarios y le den al jugador más motivación para seguir jugando, siendo como un objetivo concreto que ofrece al jugador para tener una dirección del partido.

Y, como en todo proyecto, siempre hay cosas que se pueden pulir: animaciones más suaves, efectos de sonido más variados, ajustes en la interfaz y mejoras de equilibrio entre armas y enemigos según lo que vaya probando o comenten otros.

En resumen, el juego en general tiene una buena base para seguir creciendo. Todo está montado de forma que se le pueden seguir añadiendo cosas sin tener que empezar de cero.

Capítulo 8. Bibliografía

SteamDB. (s.f.). Steam Database

URL: https://steamdb.info/

Unity 教程团队. (s.f.). 从 0 编程制作类吸血鬼幸存者游戏 [Video]. Bilibili.

URL: https://www.bilibili.com/video/BV1Sz4y1p7Tu/

Steam. (s.f.). Vampire Survivors [Página del producto].

URL: https://store.steampowered.com/app/1794680/Vampire_Survivors/

Steam. (s.f.). Brotato [Página del producto].

URL: https://store.steampowered.com/app/1942280/Brotato/

Steam. (s.f.). Kingdom Two Crowns [Game page].

URL: https://store.steampowered.com/app/701160/_/

Steam. (s.f.). The Binding of Isaac: Rebirth.

URL:https://store.steampowered.com/app/250900/The_Binding_of_Isaac_Rebirt h/

Steam. (s.f.). Celeste.

URL: https://store.steampowered.com/app/504230/Celeste/

Steam. (s.f.). Dead Cells.

URL: https://store.steampowered.com/app/588650/Dead_Cells/

Steam. (s.f.). Stardew Valley.

URL: https://store.steampowered.com/app/413150/Stardew_Valley/

Game Assets. (s.f.). Pixel Art Magic Sprite Sheet Effects [Itch.io].

URL: https://free-game-assets.itch.io/pixel-art-magic-sprite-sheet-effects

Foozlecc. (s.f.). Pixel Magic Sprite Effects [Itch.io].

URL: https://foozlecc.itch.io/pixel-magic-sprite-effects

Toffeecraft. (s.f.). UI - User Interface Mega Pack [Itch.io].

URL: https://toffeecraft.itch.io/ui-user-interface-mega-pack

Bdragon1727. (s.f.). Custom Border and Panels Menu - All Part [Itch.io].

URL: https://bdragon1727.itch.io/custom-border-and-panels-menu-all-part

Greatdocbrown. (s.f.). Coins, Gems, etc [Itch.io].

URL: https://greatdocbrown.itch.io/coins-gems-etc

Laredgames. (s.f.). Gems & Coins Free Pack [Itch.io].

URL: https://laredgames.itch.io/gems-coins-free

Pixel-boy. (s.f.). Ninja Adventure Asset Pack [Itch.io].

URL: https://pixel-boy.itch.io/ninja-adventure-asset-pack

Cainos. (s.f.). Pixel Art Top Down Basic [Itch.io].

URL: https://cainos.itch.io/pixel-art-top-down-basic

9e0. (s.f.). Witches Pack [Itch.io]. Recuperado de

URL: https://9e0.itch.io/witches-pack

Newtonsoft. (s.f.). Json.NET. Recuperado de:

URL: https://www.newtonsoft.com/json

Capítulo 9. Anexo

Para cerrar este documento, dejo a continuación el enlace al repositorio del proyecto, donde se puede encontrar el ejecutable de Magic Overdrive junto con los scripts principales desarrollados durante el proceso.

https://github.com/minwangLou/magic-overdrive

El contenido disponible permite jugar la versión actual del juego y revisar parte de los códigos implementados para este proyecto

A cualquier persona que haya llegado hasta este punto del documento, le agradezco el tiempo y el interés, y la invito a jugar y disfrutar de lo que he construido con este proyecto.