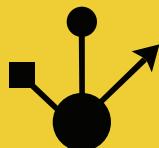




Escuela
Politécnica
Superior

Videojuego metroidvania en 2.5D



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Quico Blázquez Vidal

Tutor/es:

Alberto Real Fernández



Universitat d'Alacant
Universidad de Alicante

Videojuego metroidvania en 2.5D

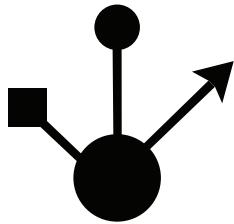
Autor

Quico Blázquez Vidal

Tutor/es

Alberto Real Fernández

Departamento de Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Multimedia



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2024

Resumen

En la última década, los videojuegos han experimentado una expansión sin precedentes, especialmente durante la pandemia de COVID-19. Dentro de este vasto mundo, el género *metroidvania* ha ganado una notable popularidad. Inspirados en clásicos como Metroid y Castlevania, estos juegos combinan la exploración no lineal con mecánicas de plataformas y acción, desafiando a los jugadores a descubrir secretos y obtener nuevas habilidades a medida que avanzan en el juego.

Este proyecto presenta el desarrollo de un prototipo, *Vertical Slice*, de un videojuego del género *metroidvania* con un estilo visual 2.5D utilizando el motor de juego Unity. El objetivo principal es desarrollar un juego que combine elementos de exploración y acción, características esenciales del género. Para ello, el trabajo se estructura en varias fases que abarcan desde el diseño conceptual hasta la implementación técnica y gráfica.

El diseño del juego incluye una mecánica principal innovadora, donde el jugador debe ejecutar habilidades trazando símbolos con un ratón, mando o tableta gráfica. Estas habilidades están asociadas a elementos como fuego, aire, agua y tierra, lo que añade profundidad y estrategia al juego.

Para la creación de enemigos, se ha desarrollado un sistema propio con un editor personalizado que facilita la generación y gestión de nuevas entidades, permitiendo una implementación rápida y eficaz de diversas estructuras y comportamientos. Este sistema utiliza técnicas de inteligencia artificial, como *Behaviour Trees* y *Utility AI*, para dotar a los NPCs de comportamientos complejos y adaptativos.

Además, se han creado editores personalizados para otros sistemas del juego, como los diálogos, mediante una estructura de nodos que permite gestionar las conversaciones de manera dinámica y flexible. El proyecto también incluye el desarrollo de menús y estados del juego, asegurando una experiencia de usuario coherente y un flujo de juego completo.

El apartado gráfico se ha centrado en ofrecer un acabado visual atractivo, empleando técnicas de post-procesado y efectos especiales como *shaders* para mejorar la inmersión del jugador. En cuanto a *sprites* o modelados, se han utilizado *assets* de terceros.

Con todo esto, este proyecto no solo aborda los aspectos técnicos y de diseño de un videojuego *metroidvania* en 2.5D, sino que también implementa soluciones innovadoras para la creación y gestión del contenido.

Resum

En la última dècada, els videojocs han experimentat una expansió sense precedents, especialment durant la pandèmia de COVID-19. Dins d'aquest vast món, el gènere *metroidvania* ha guanyat una notable popularitat. Inspirats en clàssics com Metroid i Castlevania, aquests jocs combinen l'exploració no lineal amb mecaniques de plataformes i acció, desafiant els jugadors a descobrir secrets i obtenir noves habilitats a mesura que avancen en el joc.

Aquest projecte presenta el desenvolupament d'un prototip, *Vertical Slice*, d'un videojoc del gènere *metroidvania* amb un estil visual 2.5D utilitzant el motor de joc Unity. L'objectiu principal és desenvolupar un joc que combine elements d'exploració i acció, característiques essencials del gènere. Per a això, el treball s'estructura en diverses fases que abasten des del disseny conceptual fins a la implementació tècnica i gràfica.

El disseny del joc inclou una mecànica principal innovadora, on el jugador ha d'executar habilitats traçant símbols amb un ratolí, comandament o tauleta gràfica. Aquestes habilitats estan associades a elements com foc, aire, aigua i terra, la qual cosa afegeix profunditat i estratègia al joc.

Per a la creació d'enemics, s'ha desenvolupat un sistema propi amb un editor personalitzat que facilita la generació i gestió de noves entitats, permetent una implementació ràpida i eficaç de diverses estructures i comportaments. Aquest sistema utilitza tècniques d'intel·ligència artificial, com *Behaviour Trees* i *Utility AI*, per dotar els NPCs de comportaments complexos i adaptatius.

A més, s'han creat editors personalitzats per a altres sistemes del joc, com els diàlegs, mitjançant una estructura de nodes que permet gestionar les converses de manera dinàmica i flexible. El projecte també inclou el desenvolupament de menús i estats del joc, assegurant una experiència d'usuari coherent i un flux de joc complet.

L'apartat gràfic s'ha centrat en oferir un acabat visual atractiu, emprant tècniques de post-processat i efectes especials com *shaders* per millorar la immersió del jugador. Pel que fa a *sprites* o modelats, s'han utilitzat *assets* de tercers.

Amb tot això, aquest projecte no només aborda els aspectes tècnics i de disseny d'un videojoc *metroidvania* en 2.5D, sinó que també implementa solucions innovadores per a la creació i gestió del contingut.

Abstract

In the past decade, video games have experienced unprecedented growth, especially during the COVID-19 pandemic. Within this vast world, the *metroidvania* genre has gained notable popularity. Inspired by classics such as *Metroid* and *Castlevania*, these games combine non-linear exploration with platforming and action mechanics, challenging players to discover secrets and acquire new abilities as they progress through the game.

This project presents the development of a prototype, a *Vertical Slice*, of a 2.5D visual style metroidvania genre video game using the Unity game engine. The main objective is to develop a game that combines exploration and action elements, which are essential characteristics of the genre. To achieve this, the work is structured in several phases, ranging from conceptual design to technical and graphical implementation.

The game design includes an innovative main mechanic where the player must execute abilities by tracing symbols with a mouse, controller, or graphic tablet. These abilities are associated with elements such as fire, air, water and earth, adding depth and strategy to the game.

For the creation of enemies, a custom system with a personalized editor has been developed to facilitate the generation and management of new entities, allowing for quick and efficient implementation of various structures and behaviors. This system uses artificial intelligence techniques, such as Behaviour Trees and Utility AI, to provide NPCs with complex and adaptive behaviors.

Additionally, custom editors have been created for other game systems, such as dialogues, using a node structure that allows dynamic and flexible conversation management. The project also includes the development of menus and game states, ensuring a coherent user experience and a complete game flow.

The graphical section has focused on offering an attractive visual finish, using post-processing techniques and special effects like shaders to enhance player immersion. Regarding sprites or models, third-party assets have been used.

With all this, this project not only addresses the technical and design aspects of a 2.5D metroidvania video game but also implements innovative solutions for content creation and management.

Agradecimientos

Me gustaría expresar mi más profundo agradecimiento a mi tutor, Alberto, por su dedicación, orientación y valiosos consejos. Gracias a su compromiso, he podido mantener un ritmo adecuado para la realización de este proyecto, lo que ha sido esencial para alcanzar los objetivos propuestos y superar los desafíos que se han presentado en el camino.

Quiero también agradecer a mis amigos y pareja por su apoyo incondicional y sus sabios consejos a lo largo de estos meses. La disposición para escuchar y ofrecer perspectivas diferentes ha sido fundamental para el éxito de este trabajo.

Finalmente, deseo expresar mi gratitud a mi familia por su constante apoyo a lo largo de todos estos años de carrera y por motivarme a seguir adelante y dar lo mejor de mí en cada etapa de este proceso.

Citas

Si el cajón de mis ideas no está ordenado y no tengo bien claro el objetivo que me guía en la búsqueda de una solución, no puedo tener ninguna idea.

Shigeru Miyamoto.

In my heart, I am a gamer.

Satoru Iwata.

Índice general

Resumen	v
Resum	vi
Resum	vi
Abstract	vii
Agradecimientos	ix
Citas	xi
Índice de contenidos	xiv
Índice de figuras	xvii
1 Introducción	1
1.1 Objetivos	1
1.2 Metodología	2
1.2.1 Planificación y fases del proyecto	2
1.2.2 Herramientas de planificación de trabajo	3
2 Metroidvania, 2.5D y inteligencia artificial	5
2.1 Género <i>metroidvania</i>	5
2.2 Estilo 2.5D	6
2.3 Motores de videojuegos	9
2.3.1 Unreal Engine	10
2.3.2 Unity	10
2.4 Inteligencia Artificial en Videojuegos	11
2.4.1 Enfoques para el desarrollo	12
2.4.2 Finite States Machine	12
2.4.3 Behaviour Tree	13
2.4.4 Utility AI	15
3 Desarrollo	17
3.1 Elección del motor: Unity	17
3.2 Entorno y trasfondo	18
3.3 Diseño base	18

3.4 Mecánicas y jugabilidad	20
3.4.1 Movimiento y físicas	20
3.4.1.1 Físicas Character Controller	21
3.4.1.2 Deslizar rampas	22
3.4.1.3 Movimiento del jugador	24
3.4.2 Habilidades	27
3.4.2.1 Mousepath Tracking	27
3.4.2.2 Resolución de trazados	31
3.4.2.3 Ralentizar el tiempo	33
3.4.2.4 Imbir	33
3.4.2.5 Pasiva	33
3.4.2.6 Activa	34
3.4.3 Sistema de combate	34
3.4.3.1 Fijado de enemigos	34
3.4.3.2 Bloqueos	35
3.4.3.3 Cambio personaje	36
3.5 Inteligencia Artificial	36
3.5.1 Behaviour Tree + Utility AI	36
3.5.1.1 Nodos	36
3.5.1.2 Editor personalizado	38
3.6 Enemigos	38
3.7 Personajes principales	44
3.8 Sistema de diálogos	46
3.8.1 Nodos y Dialogue Manager	47
3.8.2 Editor personalizado	47
3.9 Sistema de cinemáticas	48
3.10 Interfaz de Usuario (UI)	50
3.10.1 Diseño menús	50
3.10.2 Diseño HUD In-Game	51
3.11 Flujo del juego	54
3.11.1 Diagrama de flujo	54
3.11.2 Nivel, zonas y progresión	54
3.12 Post-procesado y efectos	55
3.12.1 Shaders	55
3.12.2 <i>Vignette y Bloom</i>	58
4 Resultados	61
5 Conclusiones	71
5.1 Aprendizaje y experiencia en el desarrollo	71
5.2 Limitaciones y posibles mejoras	71
Bibliografía	73

Índice de figuras

1.1 Tareas en Notion	3
2.1 Imagen del videojuego Interceptor	7
2.2 GIF Sonic The Hedgehog	7
2.3 Imagen de SimCity 2000	8
2.4 Imagen de Ori and The Blind Forest	8
2.5 Imagen de Live A Live	8
2.6 Imagen de Unravel	9
2.7 Imagen de Prince of Persia: The Lost Crown	9
2.8 Imagen de Super Paper Mario modo 2D	9
2.9 Imagen de Super Paper Mario modo 3D	9
2.10 Espacio de trabajo de Unreal Engine 4	10
2.11 Espacio de trabajo de Unity	11
2.12 Ejemplo FSM simple	12
2.13 Ejemplo Behaviour Tree	14
3.1 Diagrama de paquetes	19
3.2 Componente Character Controller	21
3.3 Ejemplo rampa	23
3.4 Detectores de paredes	25
3.5 Lógica del ascenso y descenso del salto	26
3.6 Sistema Trazados en Pokémon	27
3.7 Sistema Trazados en LostMagic	27
3.8 Representación gráfica de los puntos del trazado	28
3.9 Detección de los trazados	29
3.10 Representación de <i>PathNode</i>	30
3.11 Creación de los trazados	30
3.12 Predicción de posibles trazadas	31
3.13 Posibilidad de error en traza	31
3.14 Editor personalizado de trazados	32
3.15 Comprobación de trazados en habilidades registradas	32
3.16 Debug visual del rango de fijado de enemigos	35
3.17 Estados bloqueos	35
3.18 Inspector personalizado para <code>DefenseController</code>	35
3.19 Gráfica factor vida	37
3.20 Ejemplo editor de nodos personalizado IA	38
3.21 Campo de visión de los enemigos	39
3.22 Sprite del Goblin (<i>autor: Mattz Art</i>)	40
3.23 Behaviour Tree Goblin	40

3.24 Sprite de Demon (<i>autor: Mattz Art</i>)	41
3.25 Behaviour Tree Demon	41
3.26 Zona confort Demon	42
3.27 Sprite de Boss Hand (<i>autor: Gauna</i>)	42
3.28 Behaviour Tree Boss Hand	43
3.29 Predicción de los ataques de Boss Hand	43
3.30 Desplazamiento con coordenadas polares	44
3.31 Sprite de Naro (<i>autor: Mattz Art</i>)	45
3.32 Sprite del anciano (<i>autor: Mattz Art</i>)	45
3.33 Diagrama de clase Sistema Diálogos	46
3.34 Ejemplo editor editor de nodos personalizado diálogos	48
3.35 ScriptableObject de una cinemática	49
3.36 Boceto y diseño del menú principal	50
3.37 Boceto y diseño del menú de pausa	51
3.38 Boceto y diseño del menú <i>Game Over</i>	51
3.39 Boceto y diseño del HUD principal	52
3.40 Boceto y diseño del HUD diálogos	52
3.41 Boceto y diseño del HUD avisos	53
3.42 Diagrama de flujo del proyecto	54
3.43 Efecto cascada con <i>Unity ShaderGraph</i>	56
3.44 Efecto estado imbuido <i>Unity ShaderGraph</i>	56
3.45 Propiedades del <i>shader</i> imbuido	56
3.46 <i>ShaderGraph outline</i> de sprites	57
3.47 <i>ShaderGraph</i> del efecto imbuido	57
3.48 Efecto <i>Vignette</i>	58
3.49 Efecto <i>Glow</i>	58
3.50 <i>Post Processing Manager</i>	59
 4.1 Menú principal (Logo: <i>AI Logo Maker</i>)	61
4.2 Cinemática inicial	61
4.3 Inicio del nivel con controles	62
4.4 Cinemática con diálogo	62
4.5 Opciones de respuesta en diálogos	62
4.6 Fijado de enemigos	63
4.7 Efecto de daño en el jugador	63
4.8 Golpe a enemigo	64
4.9 Zona escalada	64
4.10 Zona de plataformas	65
4.11 Zona de plataformas 2	65
4.12 Obtención de la primera habilidad	66
4.13 Estado de imbuir	66
4.14 Mini puzzle para abrir una puerta	67
4.15 Setas con rebote	67
4.16 Obtención de la habilidad de salto entre paredes	68
4.17 Unión de un nuevo personaje	68

4.18 Menú de pausa	69
4.19 Habilidad activa (Fuego)	69
4.20 Habilidad pasiva (Tierra)	69
4.21 Combate contra un jefe	69
4.22 Estado <i>Game Over</i>	70

1 Introducción

En la última década, los videojuegos han experimentado una expansión sin precedentes, sobre todo en tiempo de pandemia COVID-19, gracias al entretenimiento que han dado durante esos meses de confinamiento. Esta industria es una de las más lucrativas a nivel mundial, ofreciendo una amplia variedad de experiencias, desde los clásicos juegos arcade hasta experiencias más complejas como puede ser la realidad virtual, abarcando diversos géneros, estilos y plataformas.

Detrás de cada juego se encuentra una compleja estructura de tecnología, la cual hace que sea posible tener estas experiencias: los motores de juego, la base sobre la que se construyen estos mundos virtuales. Hoy en día, los motores más utilizados son Unreal Engine y Unity, gracias a la gran cantidad de tutoriales y documentación disponible, así como de la potencia que ofrecen gracias a constantes actualizaciones.

Entre los géneros que han ganado popularidad en los últimos años, se encuentra el estilo *metroidvania*, inspirados en clásicos como Metroid y Castlevania. Estos juegos combinan la exploración no lineal con mecánicas de plataformas y acción desafiando a los jugadores a descubrir secretos y obtener nuevas habilidades mientras avanzan en el mundo.

Paralelamente, el auge de los juegos independientes, o *indies*, ha dado lugar a una explosión de creatividad y originalidad en la industria del videojuego. Estos títulos, desarrollados por equipos pequeños o incluso en solitario, exploran mecánicas fuera de lo convencional y buscan ofrecer desafíos y experiencias únicas y memorables. Juegos *indies* como Hollow Knight, Ori and the Will of the Wisps y Axiom Verge utilizan la estructura *metroidvania* y han ganado mucha popularidad entre los jugadores, destacando no solo por su diseño innovador sino también por su capacidad de atraer a una amplia audiencia con sus propuestas originales.

Con todo esto en mente, la propuesta y objetivos de este trabajo se explica a continuación.

1.1 Objetivos

Siguiendo lo que se ha comentado, el objetivo de este proyecto es crear una “*Vertical Slice*” de un juego de estilo *metroidvania* con gráficos 2.5D y una mecánica poco convencional. Este juego será desarrollado en Unity y se aprovechará la documentación oficial y diversos cursos o tutoriales para adquirir nuevas habilidades durante el desarrollo.

Es importante destacar que el objetivo principal de este proyecto es desarrollar todos los sistemas base y evaluar su funcionamiento. Una vez finalizado este, se llevará a cabo un estudio de viabilidad para posteriormente centrarse en el diseño y desarrollo del contenido.

Para llevarlo a cabo, se plantean los siguientes objetivos específicos:

- Realizar un estudio de los videojuegos que pertenecen al género *metroidvania* y aquellos que comparten mecánicas similares.
- Diseño y descripción de los aspectos principales del juego, así como personajes, mecánicas o enemigos.
- Diseño y desarrollo de las mecánicas incluyendo movimiento, interacciones y físicas.
- Implementar y ajustar la inteligencia artificial de los *NPC* (*Non Player Character*) para que den vida al mundo.
- Desarrollo de herramientas para facilitar el desarrollo de contenido del juego.

1.2 Metodología

Una vez expuestos los objetivos de este proyecto, se procede a explicar la metodología empleada para llevarlo a cabo. La metodología adoptada asegura un desarrollo estructurado, dividiendo el proyecto en fases definidas que permiten abordar cada aspecto de forma organizada.

1.2.1 Planificación y fases del proyecto

El desarrollo de este proyecto abarcará 4 fases fundamentales, cada una enfocada a aspectos específicos del juego:

- Primera fase: Diseño de concepto y desarrollo base
 - Estudio de las herramientas a utilizar y diseño del concepto del juego.
 - Desarrollo de las mecánicas principales de movimiento y combate para establecer una base sólida.
- Segunda fase: Desarrollo de NPC y Comportamientos
 - Diseño de (enemigos y aliados) con sus respectivas inteligencias artificiales.
 - Creación de comportamientos y rutinas utilizando sistemas como Behaviour Trees y Utility AI.
- Tercera fase: Flujo del juego
 - Diseño y desarrollo de niveles.
 - Creación del flujo general del juego, incluyendo menús, tutorial y sistemas de progresión.
- Cuarta fase: Arte y gráficos
 - Búsqueda y adquisición de assets gráficos necesarios.
 - Creación e implementación de arte y efectos visuales propios en caso de disponibilidad de tiempo.

En resumen, como se ve en las diferentes fases, el enfoque principal del proyecto será garantizar jugabilidad sólida y atractiva desde las primeras fases del desarrollo para más adelante priorizar el trabajo en la estética y diseño visual para complementar la experiencia del jugador.

1.2.2 Herramientas de planificación de trabajo

Para llevar a cabo el proyecto de manera eficiente, se emplearán diversas herramientas que facilitarán su desarrollo.

En primer lugar, para el diseño y la asignación de tareas, se utilizará Notion, un software de gestión de proyectos que permite una organización flexible y fácil de gestionar.

En la figura 1.1 se puede ver el funcionamiento y la distribución de las distintas tareas del proyecto. Estas tareas pueden ajustarse según imprevistos o cambios en la estructura del proyecto.

The screenshot shows two main sections of a Notion workspace:

- Tareas (Tasks):** A table view showing tasks categorized by project. The columns are 'Nombre de tarea' (Task Name), 'Estado' (Status), 'Fecha límite' (Deadline), and 'Prioridad' (Priority). Tasks include 'Bugs' (On Hold, Alta), 'Flujo de juego' (On Hold, Alta), 'Mecánicas' (Finalizada, Alta), 'Combat' (On Hold), 'Movimiento' (Finalizada), 'Diálogos' (Finalizada), 'Cinemáticas' (Finalizada), 'Tarea' (Sin empezar), 'Nivel' (En curso, Media), 'Interfaz' (Sin empezar, Media), 'Diseño HUD' (Sin empezar), 'Diseño Menús' (Sin empezar), 'IA' (En curso, Alta), 'Documentos' (En curso, Media), 'VFX' (Sin empezar, Baja), and 'Modelos' (Sin empezar, Baja).
- Combate (Combat):** A card-based view showing combat-related properties. Properties include 'Responsable' (Whispy Woods, On Hold), 'Estado' (On Hold), 'Resumen' (Empty), 'Fecha límite' (Empty), 'Proyecto' (Forest Blessing), 'Tarea principal' (Mecánicas), 'Prioridad' (Empty), 'Etiquetas' (Empty), and a list of sub-tasks: Puntos Débiles, Cambio Personajes, Elementos / Tipos, Barra Aguante, Ataques (Básico), Retroceso, and Fijar.

Figura 1.1: Tareas en Notion

Para la gestión del código se utilizará GitHub, una plataforma de desarrollo de software que proporciona control de versiones mediante Git. Esto facilitará la detección y corrección de errores a través de la creación de distintas ramas de desarrollo.

El repositorio constará principalmente de tres ramas: la rama **main**, donde se integrarán las versiones finales del proyecto; la rama **desarrollo**, en la cual se llevarán a cabo los desarrollos de los diferentes aspectos del proyecto, incluyendo sub-ramas para tareas específicas; y

finalmente, una rama **experimental**, derivada de *desarrollo*, para probar la compatibilidad de las distintas funciones del proyecto. La rama *experimental* no se fusionará con otras ramas para evitar la introducción de errores. Sólo la rama *desarrollo* puede fusionarse con *main*, garantizando así la estabilidad del proyecto.

Esta forma de gestionarlo es la aprendida en el último año de la carrera de Ingeniería Multimedia.

2 *Metroidvania*, 2.5D y inteligencia artificial

En el ámbito del desarrollo de videojuegos, la evolución de géneros, estilos visuales y motores de desarrollo desempeñan un papel fundamental en la creación de experiencias de juego únicas y atractivas. Desde juegos que innovan mediante la combinación de distintos géneros hasta aquellos que exploran enfoques visuales poco comunes para captar la atención de los jugadores, así como los motores que facilitan y ayudan a la creación de estos juegos.

Por ello, en este marco teórico se profundizará en tres aspectos clave para el proyecto: el género *metroidvania*, el estilo visual 2.5D, los motores de videojuegos más utilizados y la inteligencia artificial. En estos puntos se destacan sus definiciones, características y ejemplo, con el fin de comprender en profundidad su impacto en el desarrollo de videojuegos y cómo influye en el desarrollo del proyecto propuesto.

2.1 Género *metroidvania*

El término “**Metroidvania**” surge de la fusión de dos sagas de videojuegos clásicos: Metroid (1986) y Super Metroid (1994) por parte de Nintendo en el que se tenía que luchar contra unas criaturas en un torno 2D en un mapa grande y desplazable. Y, por otra parte, Castlevania II (1987) y Castlevania: Symphony of the Night (1997) por parte de Konami para PlayStation el cual comparte la acción y plataformas en 2D de Metroid pero adquiere un aspecto RPG más profundo con colecciónables (Wikipedia contributors, 2024b).

Estos juegos normalmente utilizan un mapa grande interconectado en el que el jugador puede explorar libremente encontrándose así con partes que no podrán acceder hasta que se adquiera un objeto o habilidad especial que permita el paso favoreciendo así la exploración. Esta habilidad u objeto, facilitará el movimiento por el mundo y/o el combate.

Se considera que Super Metroid, pulió las bases del juego plataforma no lineal de Metroid creando así el centro del estilo *metroidvania*. Sin embargo, no fue hasta Castlevania: Symphony of the Night, donde se consideró el *metroidvania* definitivo, incorporando elementos de rol de la saga The Legend of Zelda con recorrido no lineal dentro de Castlevania.

En la década de 2010, resurgió el género *metroidvania* gracias a juegos elogiados por los medios y desarrollados de forma independiente, estudios *indie*.

Las características más notables que se comparten en juegos como Metroid y Castlevania son:

- **Mapas dinámicos**

Estos juegos ofrecen a los jugadores un mapa interconectado y no lineal, el cual tiene zonas secretas o pasajes bloqueados que necesitan una herramienta o habilidad específica para acceder, fomentando así la expansión del mundo y la progresión del jugador (MasterClass, 2021).

- **Exploración**

Como dice el apartado anterior, la exploración es importante ya que los jugadores deben regresar a áreas previas para investigar y encontrar objetos para seguir avanzando (MasterClass, 2021).

- **Considerar la experiencia del jugador**

A menudo presentan zonas que requieren habilidades específicas por parte del jugador, como saltos precisos o resolución de puzzles, lo que añade un desafío adicional (MasterClass, 2021).

En los últimos años se han desarrollado alguno de los mejores juegos *metroidvania* y particularmente por desarrolladores *indie*.

- **Ori and the Blind Forest (Moon Studios)**

Un juego con un estilo visual atractivo con mecánicas fluidas en el que los jugadores exploran un mundo mágico y peligroso mientras van desbloqueando habilidades como puede ser un *dash*, etc. Que favorece muchísimo a la movilidad (MasterClass, 2021).

- **Axiom Verge (Thomas Happ Games LLC)**

Juego que ofrece la experiencia clásica de *metroidvania* con un toque moderno en el que los jugadores deben explorar un mundo alienígena mientras avanzan con una variedad de armas y habilidades únicas (MasterClass, 2021).

- **Hollow Knight (Team Cherry)**

Un mundo sin explicación ni contexto y deberás aprender a medida que avanzas en el juego y descubras secretos mediante el uso de habilidades que mejoran la jugabilidad, así como de amuletos que modifican levemente algunos aspectos para una experiencia más diferente para cada jugador según sus combinaciones (MasterClass, 2021).

- **Dead Cells (Motion Twin)**

Combinación de mecánicas *metroidvania* y *roguelike*. Los jugadores exploran un castillo en constante cambio, mientras se enfrentan a hordas de enemigos y van mejorando sus habilidades con cada partida (MasterClass, 2021).

2.2 Estilo 2.5D

Un juego 2.5D, es aquel que proviene de un punto medio entre el 2D y el 3D. No existe una definición acertada de este término ya que, en este espacio, los desarrolladores suelen mezclar técnicas visuales o jugables para crear experiencias únicas para los jugadores (Knight, 2021).

En el ámbito visual, los juegos arcade implementaban diferentes técnicas para simular un 3D en un entorno 2D debido a las limitaciones de hardware de la época mientras que hoy en día, el concepto 2.5D ha ido evolucionando hasta tener un entorno tridimensional con *sprites* bidimensionales.

La primera vez que apareció un juego 2.5D fue gracias a la compañía Taito Corporation, la cual publicó Interceptor en 1975, un *shooter* arcade en el que controlas un avión en primera persona con una mirilla en el centro en la que tenías que apuntar y disparar a los diferentes aviones que cambiarían su tamaño en función de la distancia, una ilusión creada a partir de un escalado a lo largo del eje Z, cuanto mas cerca de la cámara o reproductor, más grande se veían los aviones.



Figura 2.1: Imagen del videojuego Interceptor

Otra técnica comúnmente utilizada en los juegos 2D y que muestran sensación de profundidad es el *parallax scrolling*, que consiste en tener varias capas en el fondo y moverlas a distintas velocidades para dar esa sensación de movimiento y profundidad en la escena. Normalmente esta técnica se implementa en juegos 2D generalmente. Un ejemplo puede ser Sonic The Hedgehog en el que se puede ver como montañas, mar y el escenario del juego, se mueven a distintas velocidades (Wikipedia contributors, 2024a).

Figura 2.2: GIF Sonic The Hedgehog

El uso de la axonometría es bastante común en el desarrollo de un juego con la característica 2.5D ya que a partir de distintas proyecciones se puede dar una sensación de profundidad como puede ser el ejemplo de SimCity 2000 con una proyección isométrica.



Figura 2.3: Imagen de SimCity 2000

Todas estas técnicas eran utilizadas mayoritariamente por las limitaciones del hardware de los sistemas en los que se jugaba en aquel entonces. Hoy en día ya no es necesario el uso de estas técnicas debido a la evolución del hardware, pero es muy común en juegos *indie* el uso de estas como puede ser en *Ori and the Blind Forest*, que utiliza el *parallax scrolling* para dar sensación de profundidad al igual que el uso inteligente de sombras y luces (Anusky, 2020).



Figura 2.4: Imagen de Ori and The Blind Forest

El concepto 2.5D ha ido evolucionando y actualmente encontramos juegos muy famosos como *Live A Live* o *Octopath Traveller* en el que a partir de un escenario 3D y *sprites* 2D apuntando a la cámara, da un resultado llamativo y diferente de lo que ya se había visto.



Figura 2.5: Imagen de Live A Live

En cuanto a la jugabilidad, el término 2.5D se asocia a aquellos juegos en que gráficamente son 3D, pero su movilidad y mecánicas están limitadas a un 2D como puede ser el caso de juegos como Unravel o Prince of Persia: The Lost Crown.



Figura 2.6: Imagen de Unravel



Figura 2.7: Imagen de Prince of Persia: The Lost Crown

Por último, un ejemplo icónico de la utilización de un estilo 2.5D en la jugabilidad y en lo visual, es el juego Super Paper Mario para la consola Wii, donde los gráficos y jugabilidad van cambiando debido a la mecánica principal de poder cambiar la perspectiva 3D - 2D ofreciendo así una experiencia única y cautivadora para los jugadores.



Figura 2.8: Imagen de Super Paper Mario modo 2D



Figura 2.9: Imagen de Super Paper Mario modo 3D

2.3 Motores de videojuegos

El motor de un videojuego es la columna vertebral que impulsa tanto la creación como la ejecución de un juego. Funciona como una plataforma integral que proporciona a los desarrolladores una variedad de herramientas para diseñar, construir y dar vida a sus ideas.

Este ofrece desde características y funcionalidades gráficas y físicas hasta la gestión de audio e inteligencia artificial del proyecto ofreciéndose como un kit de herramientas completo que permite a los desarrolladores no preocuparse por los aspectos técnicos y complejos del desarrollo de un juego.

Actualmente hay muchísimos motores de videojuegos, pero se hablará de los dos motores ampliamente más utilizados en la industria que son: Unreal Engine y Unity.

2.3.1 Unreal Engine

Unreal Engine, desarrollado por la compañía Epic Games, ha sido un pilar en la industria del desarrollo de videojuegos desde su lanzamiento. Este motor no destaca solo por su capacidad para crear juegos de alta calidad, sino que también se utiliza para la producción de películas animadas y experiencias de realidad virtual.

Una de las principales ventajas de Unreal Engine es su potencia gráfica de alta calidad, que permite a los desarrolladores crear entornos visualmente impresionantes con efectos de iluminación avanzados, sombreado realista y detalles precisos. Además, cuenta con el sistema *Blueprint*, que facilita la programación mediante nodos, permitiendo a los desarrolladores crear lógica compleja sin tener que escribir un código extenso, lo que mejora la legibilidad del mismo. Sin embargo, una característica a tener en cuenta es que la curva de aprendizaje de Unreal Engine es algo más pronunciada en comparación con otros motores de juego (Sabiq, 2023).



Figura 2.10: Espacio de trabajo de Unreal Engine 4

2.3.2 Unity

Unity, creado por Unity Technologies, es uno de los motores más famosos y versátiles en la industria. Al igual que Unreal Engine, este motor no se limita solo a juegos, sino que también se utiliza para aplicaciones de realidad virtual, aplicaciones móviles o, así como en producción de películas y animaciones.

Una de las características destacables de Unity es su potencia multiplataforma, que permite a los desarrolladores crear juegos y aplicaciones para una amplia gama de plataformas, como

PC, consolas, móviles y sistemas de realidad virtual y aumentada. Además, Unity cuenta con una interfaz simple e intuitiva que facilita el trabajo de los desarrolladores y permite la creación de *scripts* de ventanas personalizadas. Otro aspecto importante es su comunidad muy activa, compuesta por numerosos desarrolladores y artistas dentro del ecosistema. Esta comunidad proporciona recursos, tutoriales y soporte, creando un entorno colaborativo que facilita el desarrollo de proyectos (Sabiq, 2023).

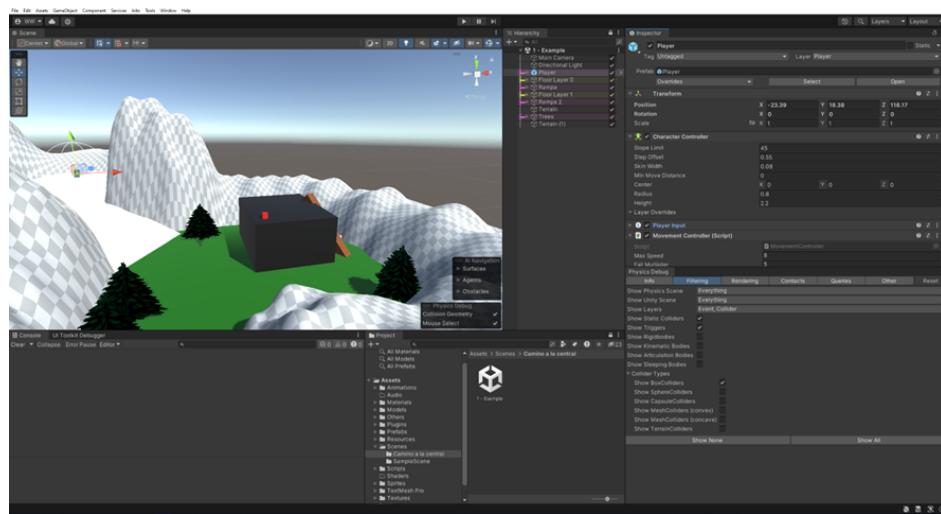


Figura 2.11: Espacio de trabajo de Unity

2.4 Inteligencia Artificial en Videojuegos

En los videojuegos, uno de los aspectos más importantes, además de los gráficos y las mecánicas, es la inteligencia artificial (IA). La IA hace que un entorno virtual se sienta vivo mediante las acciones de NPCs (*Non-Playable Characters*), enemigos, aliados y otros personajes que interactúan con el jugador. Este sistema permite simular inteligencia en las entidades y su toma de decisiones basada en parámetros del mundo y las acciones del jugador en este.

La IA en videojuegos se define como el conjunto de técnicas utilizadas para diseñar el comportamiento de los NPCs, ya sean enemigos o aliados del jugador. Estas técnicas incluyen *machine learning*, aprendizaje profundo y procesamiento del lenguaje, permitiendo que las máquinas analicen la información de su entorno y se comporten de manera similar a los humanos. Sin embargo, la aplicación de la IA en videojuegos va más allá de los NPCs y sus comportamientos, abarcando el diseño de mapeado, niveles y hasta la creación de juegos completos (Mira, 2020).

Una característica fundamental de la IA en videojuegos es su función lúdica, lo que la diferencia de la IA en otros campos donde se busca eficiencia y precisión. En videojuegos, la IA se diseña para ser entretenida y desafiante, pero no perfecta, ya que una IA excesivamente precisa podría romper la inmersión y la diversión del juego. Desde los primeros juegos como Pong y Space Invaders hasta títulos más avanzados como Metal Gear Solid, la IA ha

evolucionado considerablemente, permitiendo comportamientos más humanos y naturales en los NPCs y ofreciendo experiencias de juego más inmersas y realistas.

2.4.1 Enfoques para el desarrollo

Existen varios enfoques para desarrollar la inteligencia artificial en videojuegos, cada uno con sus propias ventajas y características. Estos métodos permiten a los NPCs tomar decisiones, adaptarse a las situaciones del juego y comportarse de maneras que mejoran la experiencia, como ya se ha comentado con anterioridad.

A continuación, exploraremos algunos de los enfoques más utilizados para implementar IA en videojuegos, analizando sus características, aplicaciones y cómo contribuyen a la jugabilidad y la inmersión.

2.4.2 Finite States Machine

Finite States Machine (FSM, por sus siglas en inglés), es un modelo matemático utilizado para representar una variedad de comportamientos dentro de un marco lógico. Este modelo se emplea en diversas aplicaciones como videojuegos, movimiento de robots, gestión de redes de comunicación, entre otros (Foquim, s.f.).

Este modelo opera a partir de un conjunto finito de estados, transiciones entre estos y un conjunto de eventos que provocan los cambios. Cada estado define una acción específica que la máquina o personaje puede realizar, como por ejemplo patrullar, perseguir, atacar, entre otras. Solo se puede procesar un único estado a la vez, lo que significa que un personaje no puede estar en dos estados simultáneamente (Król, s.f.).

Los parámetros del entorno y las acciones del jugador se envían al modelo en forma de eventos. El modelo los analiza y decide si se debe cambiar el estado actual del personaje. Cada estado tiene transiciones hacia otros estados, y estas describen las condiciones necesarias para permitir dicho cambio.

FSM Enemigo Simple

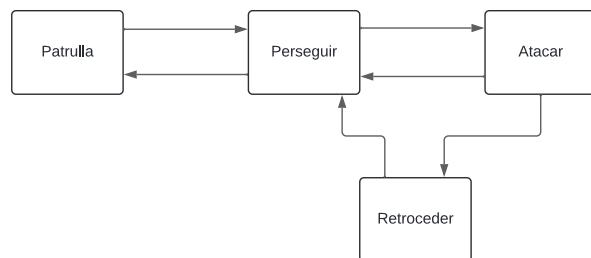


Figura 2.12: Ejemplo FSM simple

Por ejemplo, un enemigo en un juego de plataformas con los estados: patrullando, persiguiendo, atacando y retrocediendo (Figura 2.12). Cuando el enemigo está en el estado de "patrullando", se mueve de un lado a otro en un área designada. Si el jugador entra en su campo de visión, los parámetros del entorno cambian y se envía una entrada al modelo. El modelo analiza esta entrada y, si se cumplen las condiciones necesarias (por ejemplo, la distancia al jugador es menor a un cierto umbral), la FSM cambia el estado de "patrullando" a "persiguiendo".

En el estado de "persiguiendo", el enemigo sigue al jugador. Si el enemigo se acerca lo suficiente al jugador, otra condición se cumple y el estado cambia de "persiguiendo" a "atacando", donde el enemigo comienza a atacar al jugador. Si el jugador logra alejarse y salir del rango de ataque, el estado puede cambiar de "atacando" a "persiguiendo" nuevamente, o incluso a "retrocediendo" si el enemigo necesita crear distancia.

En resumen, las máquinas de estados finitos gestionan el comportamiento de los personajes en videojuegos a través de un conjunto definido de estados y transiciones entre ellos, basadas en las condiciones del entorno y las acciones del jugador.

2.4.3 Behaviour Tree

Behaviour Tree o Árbol de Comportamiento en español, es una arquitectura ampliamente utilizada en inteligencia artificial que permite a los personajes o máquinas la capacidad de seleccionar y ejecutar comportamientos mediante una estructura jerárquica en forma de árbol. Esta estructura define operaciones lógicas simples para el control del comportamiento (de Byl, 2022).

A diferencia del modelo FSM, que también utiliza un numero finito de acciones, el *Behaviour Tree* destaca por su modularidad gracias a los distintos tipos nodos que componen el árbol. Esto facilita la creación de comportamientos complejos a partir de tareas simples, permitiendo una mayor flexibilidad y adaptabilidad en la programación.

Cada nodo en un *Behavior Tree* devuelve un estado que indica el resultado de su ejecución: éxito (*success*), fracaso (*failure*) o en progreso (*running*).

Los nodos más básicos que se puede encontrar en un Behaviour Tree son:

- **Nodo Leaf (Hoja):**
 - Representa la tarea más básica dentro del árbol.
 - Ejecuta una acción específica, como moverse a una posición, disparar un arma, o interactuar con un objeto.
 - Devuelve éxito, fracaso o estado en progreso, dependiendo del resultado de la acción.
-

- **Nodo Sequence (Secuencia):**

- Organiza las acciones en una secuencia ordenada, es decir, ejecuta sus nodos hijos de forma secuencial, uno tras otro.
- Si un nodo falla, la secuencia se detiene y el nodo secuencia se marca como fracaso.
- Si todos los hijos devuelven éxito, el nodo secuencia también.
- Si alguno de los nodos está en progreso, el nodo secuencia espera a que termine para continuar.

- **Nodo Selector:**

- Similar al nodo *Sequence*, organiza acciones de una lista de nodos hijos pero con una lógica diferente.
- Ejecuta sus nodos en orden hasta que uno tenga éxito, por lo tanto, si un nodo falla, se ejecuta el siguiente.
- Si un nodo tiene éxito, el nodo Selector también.
- Si se fracasa en todas las acciones, se marcará como fracaso.
- Al igual que otros nodos, si esta un nodo hijo en progreso, se espera a que termine para poder continuar la selección.

Para entender el funcionamiento, se supone un enemigo cuyo *Behaviour Tree* sea el de la figura 2.13. Su comportamiento básico es patrullar un área. Si el NPC ve al jugador, debe seguirlo y si se acerca lo suficiente, debe atacarlo; si pierde de vista al jugador, debe volver a patrullar.

Behaviour Tree Enemigo Simple

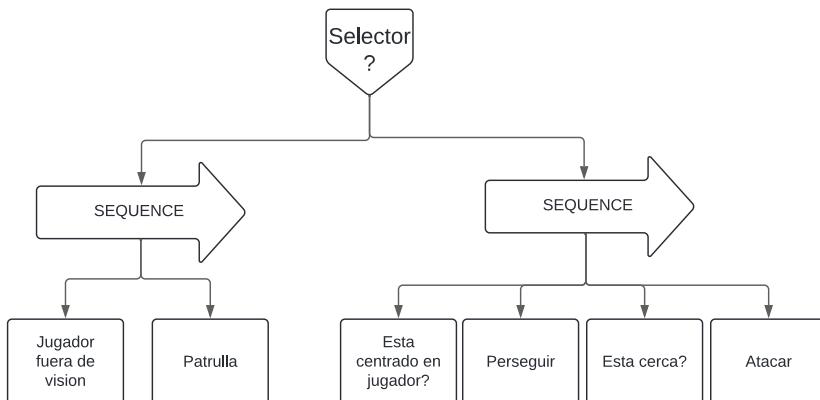


Figura 2.13: Ejemplo Behaviour Tree

El árbol comienza con un nodo Selector, que elige entre dos secuencias principales: patrullar y perseguir al jugador.

En la primera secuencia, el nodo verifica si el jugador no está visible actualmente. Si esta condición se cumple, el NPC procede a patrullar, moviéndose entre puntos predefinidos en el mapa.

En cambio, cuando el enemigo detecta al jugador, el nodo Selector cambia a la segunda secuencia. Aquí, el NPC primero verifica si está centrado en el jugador y si se cumple, inicia la acción de perseguirlo.

Durante la persecución, el árbol verifica si se está lo suficientemente cerca del jugador para realizar un ataque. Si la distancia es adecuada, ejecuta la acción de ataque.

Si en cualquier momento uno de los nodos de la secuencia falla (por ejemplo, el jugador se esconde o el NPC pierde el foco del jugador), el *Behavior Tree* se reinicia desde el principio, permitiendo volver a evaluar la situación y decidir nuevamente entre patrullar y perseguir.

Este diseño garantiza que el NPC pueda adaptarse dinámicamente a los cambios en la visibilidad del jugador y tomar las acciones apropiadas según la situación actual del juego.

2.4.4 Utility AI

Utility AI se basa en la teoría de la utilidad, una rama de la teoría de la decisión que se ocupa de la elección racional entre diferentes acciones. En este contexto, "utilidad" se refiere a una medida cuantitativa que indica como de beneficiosa es una acción en función de ciertos parámetros o condiciones. A diferencia de otros enfoques como los *Finit States Machine* o *Behaviour Tree*, que siguen reglas predefinidas, *Utility AI* calcula dinámicamente la mejor acción posible considerando múltiples factores simultáneamente.

Cada acción en una estructura de Utilidad tiene varios factores que determinan su valor final. Estos factores pueden incluir la salud actual del personaje, la distancia al objetivo, la cantidad de munición disponible, entre otros. Para obtener la puntuación final de la utilidad de una acción, estos factores pueden combinarse de diferentes maneras (Graham, 2014).

Una forma común de hacerlo es mediante la ponderación, donde se multiplica la puntuación de cada factor por su peso respectivo y luego se suman los valores. La fórmula para calcular la utilidad se puede expresar como:

$$\text{Utilidad} = \sum_{i=1}^n \text{Puntuación del Factor}_i \times \text{Peso del Factor}_i \quad (2.1)$$

Esta formula permite ajustar la importancia relativa de cada factor en la decisión final, por lo que modificando los pesos, se puede crear un mismo enemigo pero con personalidades diferentes (Graham, 2014).

Otro enfoque para el cálculo de la utilidad esperada de una acción es multiplicar la puntuación de la utilidad por la probabilidad de cada posible resultado y sumar los valores.

Para entender el concepto, se explicará un ejemplo a partir de la expresión (2.1).

Queremos calcular la utilidad de la acción "Curar", basada en dos factores: la salud actual del personaje y la distancia al jugador.

- **Factor Salud:** La puntuación del factor de la salud actual se normaliza en un valor de 0 a 1. Por ejemplo, si la salud actual es 40 % del total, entonces la puntuación sería 0.4.
- **Factor Distancia:** La puntuación del factor de distancia al jugador se puede normalizar dividiendo la distancia actual entre la distancia máxima posible. Por ejemplo, si la distancia actual es 30 unidades y la distancia máxima es 100 unidades, entonces la puntuación sería:

$$\text{Factor Distancia} = \frac{\text{distancia actual}}{\text{distancia máxima})} = \frac{30}{100} = 0.3 \quad (2.2)$$

Suponiendo que queremos que el factor de salud sea mas importante que la distancia, estableceremos como peso 1.5 y dejaremos la distancia en un peso inferior a este, por ejemplo 1. Por lo tanto siguiendo la fórmula (2.1):

$$\text{Utilidad} = (0.4 \times 1.5) + (0.3 \times 1) = 0.9 \quad (2.3)$$

Este valor se utilizaría para comparar con otras acciones y sus utilidades y se ejecutaría la acción con mayor puntuación pero también puede darse el caso de que se elija de otra forma según el usuario.

Una forma de calcular la utilidad además de el sumatorio ponderado, existen otros métodos para combinar los factores y calcular la utilidad:

- **Media Aritmética:** Realizar la media de las puntuaciones de los factores.
- **Producto:** Multiplicar las puntuaciones de los factores.
- **Máximo / Mínimo:** Usar el valor máximo o mínimo de los factores, dependiendo de la situación.

Uno de los juegos más conocidos que utiliza el sistema de *Utility AI* es "Los Sims", donde cada personaje tiene sus propias prioridades y tareas basadas en diferentes factores como el hambre, entretenimiento y otros aspectos de la vida diaria. Este enfoque permite simular comportamientos realistas y coherentes, determinando qué tipo de comida prefieren o qué actividades realizar cuando están aburridos (TheShaggyDev, 2023).

La implementación de *Utility AI* facilita el desarrollo de comportamientos de personajes con prioridades específicas según diferentes condiciones, ofreciendo una flexibilidad y capacidad de adaptación que hacen de esta técnica una herramienta poderosa para la toma de decisiones. Gracias a su capacidad para calcular dinámicamente la mejor acción posible en cada situación, *Utility AI* contribuye significativamente a mejorar la inteligencia y el realismo de los personajes en videojuegos.

3 Desarrollo

En esta sección se describirá el proceso de diseño y creación del proyecto, abarcando desde la selección del motor de juego y como se estructurará, hasta el diseño y desarrollo de las mecánicas, jugabilidad y inteligencia artificial. En cuanto al apartado artístico y diseño de nivel, se tendrá en cuenta una vez esté desarrollado.

3.1 Elección del motor: Unity

Para el desarrollo del proyecto se ha elegido Unity como motor principal. Esta decisión se ha basado en varios factores clave que hacen que sea la mejor opción para este desarrollo.

- **Flexibilidad y Soporte para 2D y 3D**

Es un motor versátil en el que ofrece soporte tanto para juegos 2D como 3D. Esta capacidad híbrida es ideal para este proyecto enfocado en el 2.5D, que combina elementos bidimensionales y tridimensionales.

- **Comunidad y Recursos**

Unity cuenta con una de las comunidades más grandes y activas en el ámbito del desarrollo de videojuegos. Esta comunidad ofrece innumerables recursos, desde foros de discusión para la resolución de problemas, hasta plugins y assets disponibles en la Unity Asset Store. Este ecosistema puede acelerar significativamente el desarrollo y resolver problemas de manera más eficiente.

- **Facilidad de uso**

Gracias a la interfaz de usuario fácil e intuitiva, lo que hace sencillo la adaptación al entorno de trabajo. Además, dispone de una amplia variedad de herramientas que facilitan el desarrollo de tareas comunes, como la física, las animaciones o la interfaz de usuario.

- **Requisitos técnicos y rendimiento**

Unity es accesible y optimizado para desarrollos que no demandan un rendimiento gráfico muy grande. Esto es particularmente beneficioso para el proyecto, ya que el juego no requiere la potencia gráfica de motores como Unreal Engine.

- **Multiplataforma**

Este motor soporta la creación de juegos multiplataforma, permitiendo desarrollar y desplegar un juego en una amplia gama de dispositivos, incluyendo PC, consolas, dispositivos móviles y realidad virtual. En este caso se enfocará en los dos primeros.

En conclusión, Unity ha sido elegido por su flexibilidad, comunidad activa, facilidad de uso, requisitos técnicos accesibles y soporte multiplataforma. Uno de los puntos más fuertes de Unity es la capacidad de crear editores personalizados, lo cual facilita aún más el desarrollo del juego y permite una adaptación específica a las necesidades del proyecto.

3.2 Entorno y trasfondo

En esta sección se establecerá el contexto del juego, proporcionando un resumen de las mecánicas, el entorno y la experiencia que se busca ofrecer.

El proyecto consiste en desarrollar un videojuego en 2.5D, inspirado en el estilo visual de juegos como *Octopath Traveler* y *Live a Live*, y una movilidad similar a *Hollow Knight*, previamente comentados. Estos juegos destacan por su uso de la iluminación y la combinación de sprites 2D en entornos 3D.

La mecánica principal será un sistema de combate basado en trazado, donde no solo se combate pulsando una tecla, sino que también se considera la dirección del ataque. Esta mecánica será fundamental y estará integrada con elementos que favorecerán la exploración de niveles, similar a los juegos de estilo metroidvania.

Para garantizar una variedad de habilidades, cada una estará asociada a un elemento (Fuego, Aire, Agua, Tierra). Esto permitirá crear varios elementos en el escenario que requieran una habilidad específica para acceder a ciertas áreas o incluso para enfrentar a un jefe, donde será necesario usar una habilidad de un elemento en específico para poder dañarlo en ciertos momentos del combate.

En cuanto a los enemigos, estos podrán defenderse, lo que obligará al jugador a pensar estratégicamente sobre la mejor dirección para atacar. Si el jugador comete un error, quedará paralizado temporalmente, añadiendo una capa adicional de desafío.

Como se ha mencionado, habrá varios tipos elementales, lo que permitirá tener un jugador para cada elemento. Esto favorecerá el cambio de personaje y el uso de distintas habilidades durante el combate, enriqueciendo la jugabilidad y la estrategia en cada enfrentamiento.

Una vez esté la base desarrollada, se crearán sistemas para la gestión de diálogos y cinemáticas ya que se quiere contar una historia con variaciones en su experiencia.

Por lo que este proyecto se centra en crear una "Vertical Slice" que abarque lo comentado.

3.3 Diseño base

Antes de comenzar el desarrollo, se creará de forma preliminar un diagrama de paquetes para definir cómo queremos que funcione el juego a nivel técnico y gestionar la estructura de manera clara y organizada. Este diagrama servirá como guía principal para asegurar que

los aspectos básicos del juego estén bien estructurados y que el desarrollo siga un enfoque coherente.

DIAGRAMA DE PAQUETES

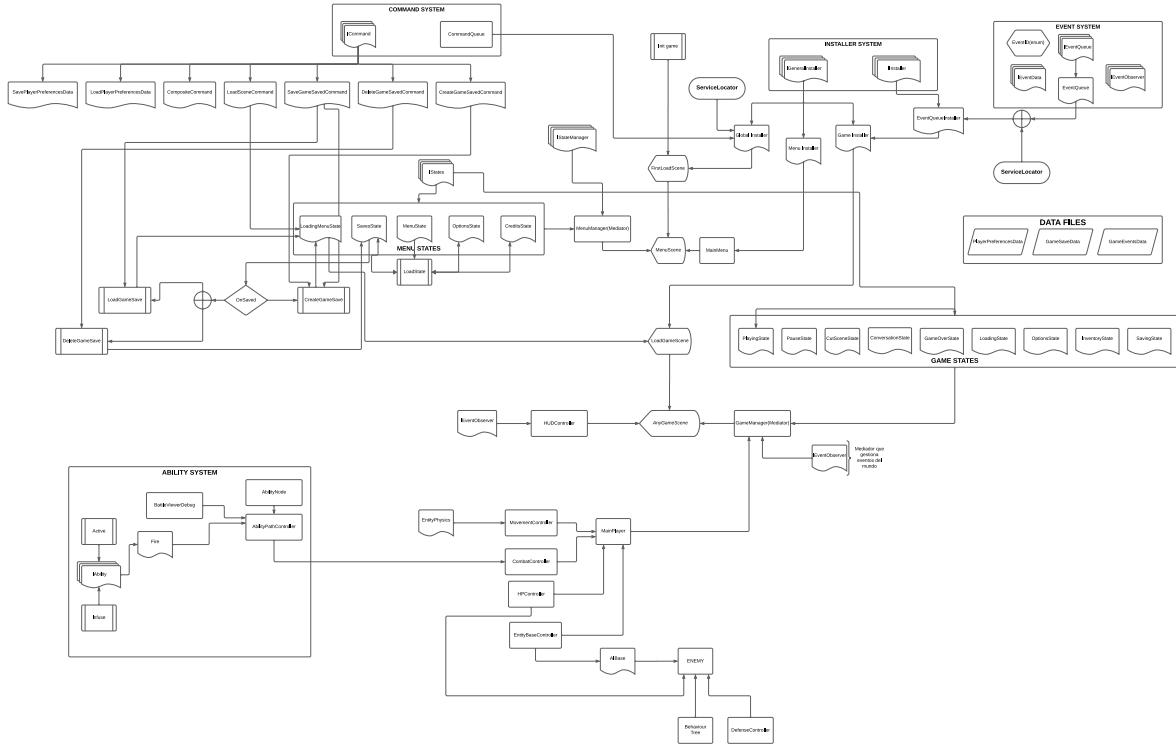


Figura 3.1: Diagrama de paquetes

En el diagrama de la figura 3.1 proporciona una primera visión de la estructura que seguirá el proyecto. A continuación, se explica cada sistema del diagrama de manera ordenada.

- **Command System**

Este sistema se encarga de gestionar los comandos de diversas acciones del juego, como guardar, cargar niveles, y crear partidas, entre otros. Se basa en el patrón de diseño *Command*, permitiendo desacoplar las solicitudes de las acciones que las ejecutan (Parra, 2023).

- **Installer System**

Responsable de la instalación y configuración de los distintos sistemas del juego, asegurando la correcta resolución de dependencias. Este sistema permite inicializar componentes esenciales del juego de manera ordenada (Parra, 2023).

- **Event System**

Los eventos son fundamentales para la comunicación entre diferentes componentes del juego. Este sistema se basa en el patrón *Observer* y utiliza una cola (*Queue*) para

manejar múltiples eventos en un mismo frame. Además, se implementa siguiendo el patrón *ServiceLocator*, lo que permite un acceso escalable y centralizado a los servicios (Parra, 2023).

- **Menu States**

Basado en el patrón *State* (Parra, 2023), este sistema gestiona los diferentes estados del menú del juego, como los menús de carga, opciones y créditos.

- **Game States**

Similar al sistema de estados del menú, este también está basado en el patrón *State*, pero se enfoca en gestionar los diferentes estados del juego, como el estado de juego, pausa, diálogo y más.

- **Ability System**

Este sistema gestiona las habilidades y poderes del jugador. Estas habilidades son cruciales tanto para avanzar a nuevas áreas como para combatir enemigos. Este sistema será el encargado de manejar el sistema de trazado y la resolución de éste.

- **Componentes adicionales**

- **HUDController:** Maneja los eventos de la interfaz de usuario, realizando las animaciones y cambios necesarios para reflejar el estado del juego.
- Las entidades, como enemigos o el jugador, cuentan con componentes específicos para la movilidad y la gestión de la salud, asegurando un comportamiento coherente y dinámico dentro del juego.

3.4 Mecánicas y jugabilidad

En esta sección, se detallarán las principales mecánicas y elementos de jugabilidad que definirán la experiencia del jugador. La jugabilidad de un videojuego es el aspecto más crucial, ya que determina cómo los jugadores interactúan con el mundo del juego, qué acciones pueden realizar, y cómo estas acciones afectan su progreso. A continuación, se desglosan las diversas mecánicas que componen el sistema de movimiento, las físicas del juego, el sistema de combate, y otras habilidades especiales que estarán a disposición del jugador.

3.4.1 Movimiento y físicas

Las mecánicas de movimiento y físicas son fundamentales para garantizar una experiencia de juego fluida y atractiva. Juegos como Hollow Knight y Celeste destacan por su excelente movilidad y físicas, ofreciendo una jugabilidad cómoda y precisa. En esta sección, se describen los componentes clave necesarios para lograr un comportamiento similar para el proyecto, incluyendo el controlador de personajes, las mecánicas de salto, y la interacción física con el entorno.

3.4.1.1 Físicas Character Controller

En Unity, existen dos formas principales de programar el movimiento de un objeto: mediante el sistema de físicas de Unity utilizando el componente *RigidBody*, o mediante el componente *Character Controller*, que no está ligado a estas físicas realistas.

La elección entre un componente u otro es crucial. Si se busca un movimiento sencillo y realista, *RigidBody* es la mejor opción, ya que proporciona métodos y sistemas para manejar diferentes tipos de fuerzas y físicas. Por otro lado, si se desea un comportamiento más detallado y que no esté restringido por las físicas realistas, *Character Controller* es una mejor opción. Este componente soluciona colisiones simples, como rampas o escaleras, y permite crear una movilidad personalizada mediante la programación de un sistema de físicas propio.

Inicialmente, se desarrolló con un *RigidBody*, pero al intentar crear un salto similar a los juegos mencionados anteriormente, la lógica se complicó considerablemente. Por lo tanto, se optó por implementar un sistema de físicas personalizado para los personajes.

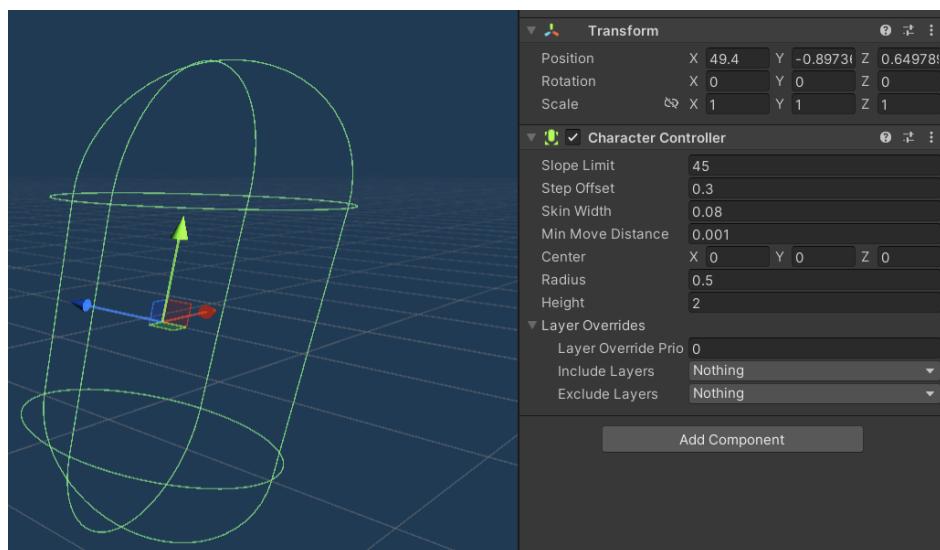


Figura 3.2: Componente Character Controller

Por lo tanto, es necesario crear un *script* para implementar las físicas personalizadas en los objetos que requieran un comportamiento específico. Para ello, desarrollaremos el *script* **EntityPhysics**, el cual se menciona en el diagrama de la figura 3.1.

Para el uso de este componente, se requiere que el objeto tenga un componente *Character Controller* para manejar el movimiento y colisiones. El *script* **EntityPhysics** inicializa diversas propiedades relacionadas con la física como velocidad máxima, gravedad, fricción, y otras específicas para el comportamiento en pendientes y bajo el agua.

El manejo del movimiento y las físicas se realiza de la siguiente manera:

- **Gravedad:** Se ajusta la gravedad basándose en si la entidad está en el aire o en el suelo. En casos como colisión con un objeto elástico o con una rampa muy pronunciada, la gravedad puede variar para conseguir el resultado adecuado a cada caso.
- **Control de Velocidad:** Se limita la velocidad de la entidad para asegurarse de que no exceda una velocidad máxima definida, ajustando según la fricción del suelo. Esto permite que al acelerar o frenar, haya un límite y no pueda salir disparado el objeto.
- **Detección de los elementos:** Para llevar a cabo muchas de las funciones físicas, es fundamental contar con el elemento o entidad externa que proporcione la información necesaria, como la cantidad de rebote o deslizamiento del personaje. El **CharacterController** permite detectar colisiones y proporciona información adicional, como la normal de la colisión, entre otros datos.

Para el cálculo del rebote, se utiliza el concepto de tiro vertical o lanzamiento vertical, un movimiento rectilíneo uniformemente acelerado (M.R.U.A.). Este se utiliza para obtener qué altura debería conseguir el personaje en un rebote y así manejar en un caso a parte la velocidad a la que queremos que suba. A continuación se explica el concepto matemáticamente:

La posición y en cualquier tiempo t

$$\mathbf{y} = y_o + V_o \times t - \frac{(g \times t^2)}{2} \quad (3.1)$$

Velocidad final V_f

$$V_f = V_o + g \times t \quad (3.2)$$

En el punto más alto, $V_f = 0$

$$V_f = 0 \Rightarrow 0 = V_o + g \times t \Rightarrow t = \frac{V_o}{g} \quad (3.3)$$

Por tanto, la altura máxima se puede expresar como:

$$\boxed{\maxHeight = y_o + \frac{V_o^2}{2g}} \quad (3.4)$$

3.4.1.2 Deslizar rampas

El movimiento en terrenos inclinados se ajusta para restringir áreas o evitar que el jugador pueda volver por el mismo camino. Este código permite controlar la velocidad de deslizamiento y la inclinación necesaria para activarlo.

Cuando se detecta una colisión o se realiza un *raycast* en Unity, se puede obtener el vector normal \vec{N} , el cual se utiliza para calcular la inclinación de la pendiente. Esto se puede hacer de dos maneras: calculando el ángulo entre \vec{N} y el vector horizontal, o a partir de la figura 3.3. Se ha decidido la primera opción debido a que no es necesario comprobar el signo del ángulo y por tanto es más sencillo la lógica.

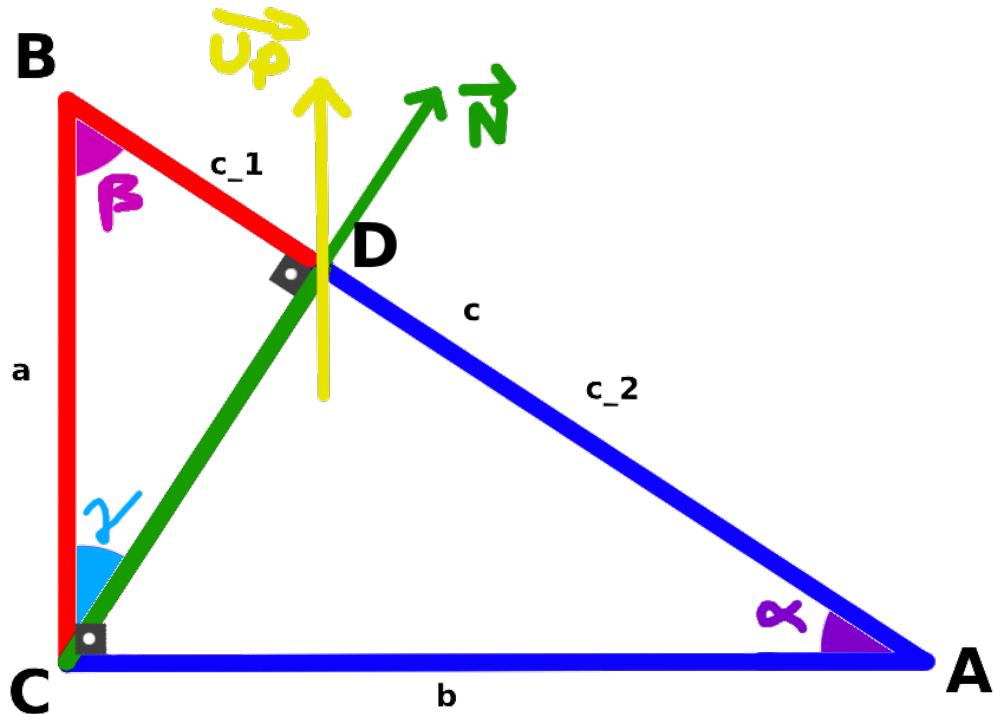


Figura 3.3: Ejemplo rampa

Según esta figura 3.3, el ángulo de inclinación de la rampa ($\angle BAC$) es igual al ángulo formado por el vector \vec{N} y el vector vertical $\vec{U}p$ ($\angle BCD$).

Esto se debe a que los triángulos $\triangle ABC$ y $\triangle DBC$ comparten dos ángulos iguales: el ángulo recto 90° ($\angle BCA$ y $\angle BDC$) y el ángulo β ($\angle DBC$). Dado que la suma de los ángulos internos de un triángulo es 180° , se puede usar un sistema de ecuaciones para demostrar que el ángulo de inclinación de la rampa es igual al ángulo formado por \vec{N} y $\vec{U}p$, que coincide con γ .

En el triángulo $\triangle ABC$:

$$\alpha + \beta + 90 = 180 \quad (3.5)$$

En el triángulo $\triangle DBC$:

$$\gamma + \beta + 90 = 180 \quad (3.6)$$

Ambos triángulos comparten el ángulo β y el ángulo recto (90°), se puede deducir que:

$$\boxed{\alpha = \gamma} \quad (3.7)$$

Por lo tanto, el ángulo de inclinación de la rampa ($\angle BAC$) es igual al ángulo formado por \vec{N} y $\vec{U}p$ ($\angle BCD$).

3.4.1.3 Movimiento del jugador

Basándose en `EntityPhysics`, se crea un *script* que hereda de este, llamado `MovementController`, el cual añade y modifica funciones para el movimiento específico del jugador. Este componente incorpora la lógica necesaria para controlar las acciones del jugador, aprovechando el sistema de física base.

Antes de explicar las funcionalidades específicas, es fundamental entender la detección de input, que permite que el personaje responda a las diferentes acciones. Para ello, se utiliza el nuevo sistema de input de Unity. Este sistema se basa en un mapa de entrada, donde se definen los controles para diversas acciones, ya sea con teclado, mando u otros dispositivos. Posteriormente, estas entradas se asignan a su funcionalidad en el código, permitiendo un desacoplamiento del input y facilitando cambios en los controles de manera sencilla.

Las funcionalidades del `MovementController` incluyen:

- **Movimiento:** Permite al jugador moverse en un espacio 3D. La entrada de movimiento se captura a través del sistema de input y se transforma en un vector que influye en la dirección y la velocidad del personaje. El movimiento también considera la fricción del suelo para ajustar la aceleración.
- **Dash:** Proporciona la habilidad de realizar un deslizamiento rápido en la dirección del movimiento. Este impulso es de corta duración y está limitado por un temporizador de enfriamiento, asegurando que el jugador no pueda usar el *dash* de manera continua. El objetivo de esta función es permitir movimientos rápidos y esquivar con facilidad.
- **Salto:** Similar al sistema de salto en Hollow Knight, este permite que la altura del salto dependa del tiempo que se mantiene presionada la acción de saltar. Una pulsación rápida resulta en un salto más bajo, mientras que mantenerla presionada permite un salto más alto.

Para mejorar la jugabilidad y evitar saltos injustos, se implementa *Coyote Time*, un sistema que permite al jugador saltar poco después de haber dejado una plataforma. Esto se logra mediante un *raycast* que detecta el suelo; al perder contacto, se inicia un contador que permite saltar durante un breve período de tiempo.

Para prevenir frustraciones por parte del jugador en momentos en los que necesite saltar continuamente, se implementa un *buffer de salto*, una lógica que permite almacenar la acción de saltar antes de tocar el suelo. Esto asegura que los saltos se registren incluso si el jugador presiona el botón ligeramente antes de aterrizar.

- **Salto en pared:** Basándose en la lógica del salto normal, se implementa la capacidad de realizar saltos en paredes. Esto amplía las posibilidades de exploración y diseño de niveles, típico de los juegos tipo *metroidvania*.

Para detectar cuando el jugador toca una pared, se utilizan dos esferas a los laterales del personaje como detectores (Figura 3.4).

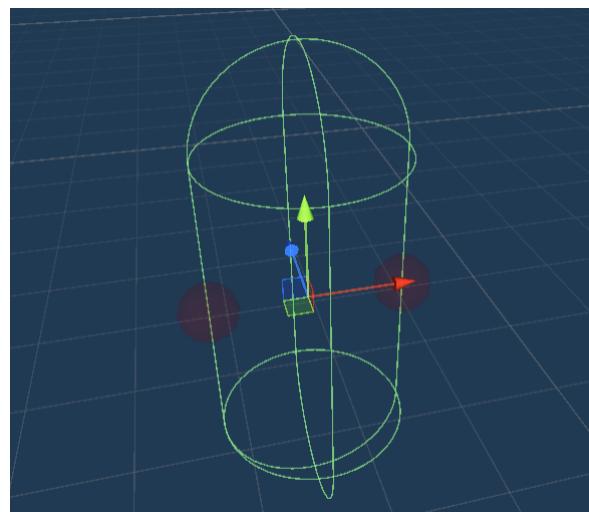


Figura 3.4: Detectores de paredes

Para que no se pueda aprovechar de manera no controlada el salto entre paredes, no se pueden realizar saltos consecutivos en la misma pared, requiriendo al jugador moverse entre dos paredes para trepar.

La lógica del salto en pared incorpora un margen de tiempo entre saltos para evitar interrupciones mientras el jugador está en el aire. Además, se implementa una dirección diagonal en el salto para que no sea completamente vertical, facilitando al jugador el cambio de pared durante la escalada.

Además para lograr un movimiento rápido y personalizado, se implementa una lógica que controla la velocidad de subida y bajada del salto, similar a la utilizada en los juegos referentes como Hollow Knight.

Esta lógica utiliza una variable que determina la duración del salto, junto con un contador el cual registra el tiempo mientras se presiona la acción de saltar. Se calcula la relación entre el tiempo transcurrido y la duración total del salto para que, cuando se haya alcanzado la mitad del tiempo del salto, el multiplicador de subida comience a disminuir en función de la inversa de esta relación ($1 - \text{relación}$).

De esta manera, la velocidad de ascenso disminuye progresivamente. Si el salto se cancela antes de lo previsto, se maneja con otra lógica para que el descenso sea rápido.

```
1  Jump();
2  // Jump speed
3  if (movement.y > 0 && IsJumping)
4  {
5      _jumpCounter += Time.deltaTime;
6
7      // LIMIT TIME
8      if (_jumpCounter > _maxJumpTime) IsJumping = false;
9
10     float jumpRelation = _jumpCounter / _maxJumpTime;
11     float currentJump = upMovementMultiplier;
12
13     // Halfway through, It starts to descend
14     if (jumpRelation > 0.5f)
15     {
16         // CurrentJump get a smooth movement
17         currentJump = upMovementMultiplier * (1 - jumpRelation);
18     }
19     movement.y += Physics.gravity.y * currentJump * Time.deltaTime;
20
21     _animator.SetTrigger("Jump");
22 }
```

Figura 3.5: Lógica del ascenso y descenso del salto

Con esta implementación (Figura 3.5), cuanto más alto quiera saltar el jugador, más tiempo se mantendrá en el aire.

3.4.2 Habilidades

Antes de desarrollar el combate del juego, es necesario implementar la mecánica de detección de trazados y resolución de símbolos para las habilidades.

Para la creación de este sistema, se ha tomado como referencia juegos como Pokémon Ranger: Trazos de luz y LostMagic, juegos de Nintendo DS, donde las habilidades o invocación se ejecutan mediante trazados en la pantalla.



Figura 3.6: Sistema Trazados en Pokémon



Figura 3.7: Sistema Trazados en LostMagic

En el caso de este proyecto, se busca permitir la realización de trazados sin la necesidad de apuntar a zonas específicas ni de replicar con exactitud la forma referente, haciéndolo de manera intuitiva y basada en direcciones rectas. Este enfoque crea una mecánica diferente pero inspirada en el mismo sistema de estos juegos, utilizando trazados compuestos por 8 posibles puntos.

Con el uso de trazados y direcciones rectas, el juego se puede jugar con teclado y ratón, tableta gráfica, o incluso con un mando utilizando su joystick, permitiendo jugar con diversos tipos de hardware sin necesidad de pulsar una pantalla ni nada parecido.

3.4.2.1 Mousepath Tracking

Teniendo como referencia el sistema utilizado por Pokémon, la idea es que las direcciones rectas trazadas por el jugador con el ratón o joystick, permitan realizar tanto trazados simples de una única dirección o símbolos complejos.

Este sistema se diseñó así, pensando en la posibilidad de jugar en cualquier plataforma y en cualquier dispositivo de entrada.

Como ya se ha mencionado, se debe pasar por 8 puntos: 2 vertical, 2 horizontal y 4 en diagonal (8 puntos siguiendo la forma de un círculo), similar al funcionamiento del sistema de Pokémon. Para representar estos nodos, se debe crear una plantilla que asegure que los trazados pasen por puntos predefinidos de manera interna.

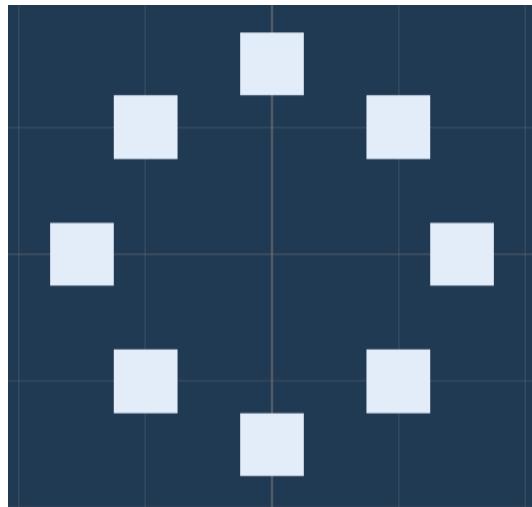


Figura 3.8: Representación gráfica de los puntos del trazado

Como se muestra en la figura 3.8, la forma es similar a la que utiliza el juego de referencia, aunque el funcionamiento es totalmente distinto.

Primero se obtiene la dirección del trazado del ratón. Unity permite obtener la dirección delta del movimiento de un ratón gracias a su nuevo sistema de entrada.

Para mejorar la precisión y evitar detecciones no intencionadas, se establece una "zona muerta" (véase la figura 3.29), que ignora los movimientos pequeños. Con un mando, el proceso es el mismo, registrando la entrada delta de un joystick. Una vez implementada la lógica de entrada, se crea el sistema.

Este sistema no debe funcionar constantemente, debe activarse cuando el jugador quiere realizar un trazado o ejecutar un ataque. Es por eso que se necesita una entrada adicional, en el caso del ratón, el clic izquierdo. Mientras se mantiene pulsado, el sistema lee las direcciones, y al soltar, ejecuta la acción correspondiente.

Durante el trazado, el jugador no puede moverse, manteniéndose inmóvil mientras realiza el trazado y se lee la dirección la cual se envía al controlador de habilidades (`AbilityPathController`).

```

1  private IEnumerator AddMovementDirection()
2  {
3      // Wait until MouseRecording has finished
4      yield return StartCoroutine(MouseDeltaRecording());
5
6      if (_directionDelta != Vector2.zero)
7      {
8          Vector2 currentDirection =
9              _abilityPathController.GetDirection(_directionDelta);
10         if (_lastDirection != currentDirection &&
11             _lastDirection != -currentDirection)
12         {
13             _lastDirection = currentDirection;
14             _abilityPathController.AddDirection(_directionDelta);
15             numDirectionsAttack++;
16
17             if (numDirectionsAttack == 7)
18             {
19                 // Execute Attack because is impossible to add more points
20                 ExecuteAttack(new InputAction.CallbackContext());
21             }
22         }
23     }
}

```

```

1  private IEnumerator MouseDeltaRecording()
2  {
3      float initTime = Time.time;
4      IsDeltaRecording = true;
5      _directionDelta = Vector2.zero;
6
7      while (Time.time - initTime <
8          inputData.MaxTimeXTrail)
9      {
10         _directionDelta += _pointerDelta;
11         yield return new
12             WaitForSeconds(_timeBetweenReads);
13     }
14
15     _directionDelta.Normalize();
16     IsDeltaRecording = false;
}

```

Figura 3.9: Detección de los trazados

Como se puede ver en el código de la figura 3.9, se establece una segunda corutina que se encarga de leer un numero de direcciones durante un tiempo dado y obtiene su media, ya que es difícil realizar trazados de lineas rectas con un ratón sin poder ver su traza.

Las direcciones se envían al controlador de habilidades, que devuelve la dirección más cercana al punto deseado por el jugador. Por ejemplo, si el jugador quiere ir desde el nodo $(-1, 0)$ al nodo $(1, 0)$, la dirección debería ser de $\vec{D} = (1, 0)$. El controlador se encarga de identificar el nodo más cercano a la dirección delta del jugador (Línea 8 Figura 3.9). Si esta dirección resultante es la misma o la inversa, no se almacena, ya que son casos imposibles de trazar.

Para realizar estas comprobaciones y obtener el nodo más cercano, se debe tener un objeto que almacene los posibles recorridos desde cada nodo. Por ello, se crea una clase llamada *PathNode*, que contiene la posición del nodo en un espacio local 2D y una lista de todos los demás nodos y sus direcciones a estos con respecto a su posición, como se muestra en la figura (Figura 3.10).

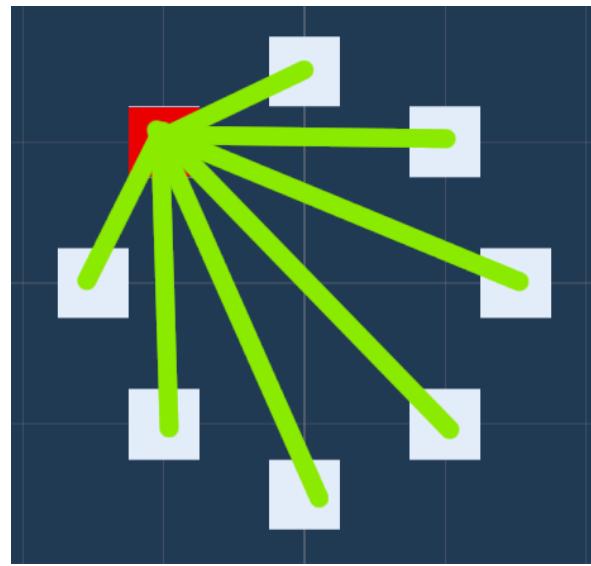


Figura 3.10: Representación de *PathNode*

Una vez se tiene la base y el cálculo correcto para las direcciones, se procede a crear los trazados a partir de estas.

```

1  public void AddDirection(Vector2 direction)
2  {
3      List<List<PathNode>> newPossibleStructures = new();
4
5      if (_possiblePaths.Count == 0)
6      {
7          GetFirstPath(direction);
8      }
9      else
10     {
11         foreach (var possiblePath in _possiblePaths)
12         {
13             var lastNode = possiblePath[possiblePath.Count - 1];
14             foreach (var childNode in lastNode.childNodes)
15             {
16                 if (possiblePath.Exists(nodes => nodes.Equals(childNode.Item1)))
17                     continue;
18
19                 float angle = Vector2.Angle(childNode.Item2, direction);
20                 if (angle < 23.0f)
21                 {
22                     List<PathNode> newPath = new List<PathNode>(possiblePath)
23                     {
24                         childNode.Item1
25                     };
26                     newPath.Add(newPath);
27                 }
28             }
29         }
30         _possiblePaths = newPossibleStructures;
31     }
32 }
```

Figura 3.11: Creación de los trazados

El código (Figura 3.11), crea una Lista de posibles trazados que el jugador podría realizar. Si no hay posibles caminos, la primera dirección establece el nodo de salida y su siguiente nodo según la dirección. Si se obtiene la dirección $\vec{D} = (1, 0)$, hay 3 posibles soluciones que coinciden con ese parámetro (véase la figura 3.17).

A medida que se añaden direcciones, se comprueba si la siguiente dirección existe desde el último nodo de cada posible camino. Si existe, la estructura se mantiene con el nuevo nodo, y las que no coincidan se descartan.

El cálculo del ángulo (línea 21), permite almacenar dos direcciones muy similares, como en la figura 3.13. La linea amarilla indica la siguiente dirección desde el punto rojo, pero no se puede saber si el jugador quería el nodo trazado por la amarilla o el nodo trazado por la verde, por lo que se añaden ambas posibilidades.

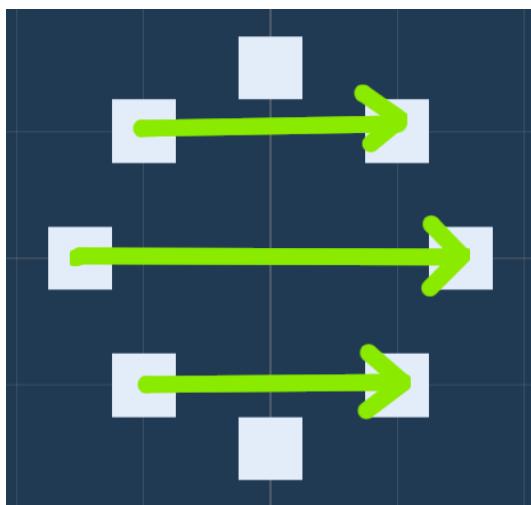


Figura 3.12: Predicción de posibles trazadas

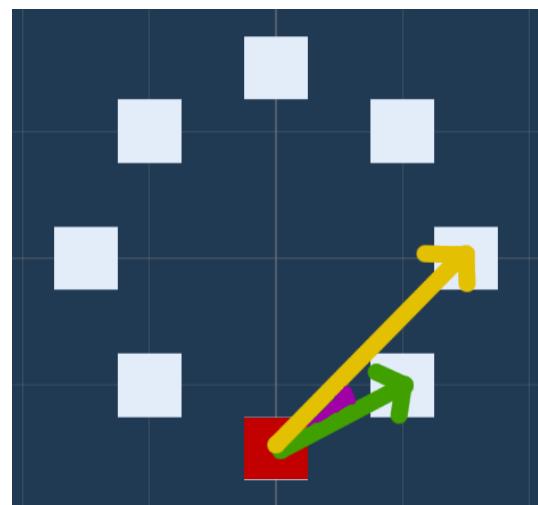


Figura 3.13: Posibilidad de error en trazada

En conclusión, a medida que se añaden direcciones, se almacena los posibles trazados o caminos que el jugador intenta realizar.

La siguiente sección explica cómo se obtiene el trazado correcto.

3.4.2.2 Resolución de trazados

Para verificar qué trazado intenta hacer el jugador, es necesario crear un objeto que almacene una lista de habilidades en el juego. Es por eso que se creará un *ScriptableObject* que contendrá la información necesaria del trazado de una habilidad, así como los puntos que recorre.

Para facilitar la creación de estas habilidades, se ha creado un editor personalizado en Unity usando *UIBuilder* para la interfaz visual y un script para su funcionamiento, permitiendo dibujar el trazado desde el inspector. (Figura 3.14).

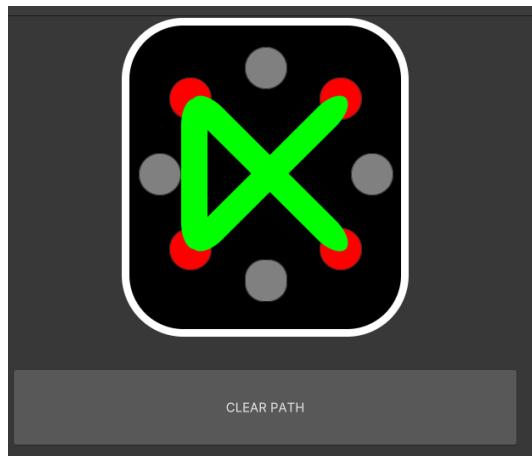


Figura 3.14: Editor personalizado de trazados

Este *ScriptableObject* almacena una lista de nodos en el orden en el que fueron dibujados. Sin embargo, surge un problema: ¿qué pasaría si el jugador realiza el trazado a la inversa?

Para solucionar esto, se comprueba el primer nodo de la habilidad almacenada y en qué estructuras empieza o termina. Una vez identificado un posible camino, se recorren todos sus nodos en el orden correspondiente y, si se llega al final sin errores, se obtiene la habilidad correcta (Figura 3.15).

```

1  public string GetAbilityName()
2  {
3      float marginError = 0.1f;
4      // Search the correct Ability. If game runs slow, program with async or coroutine
5      for (int index = 0; index < _abilityPaths.Count; index++)
6      {
7          var ability = _abilityPaths[index];
8          foreach (var possiblePath in _possiblePaths)
9          {
10              if (possiblePath.Count != ability.Count) continue;
11              if (Vector2.Distance(possiblePath[0].Position, ability[0]) <= marginError)
12              {
13                  for (int i = 1; i < possiblePath.Count; i++)
14                  {
15                      if (Vector2.Distance(possiblePath[i].Position, ability[i]) > marginError) break;
16                      // FinalPath
17                      if (i == possiblePath.Count - 1)
18                          return _abilityPaths[index].AbilityPathName;
19                  }
20              }
21              else if (Vector2.Distance(possiblePath[^1].Position, ability[0]) <= marginError)
22              {
23                  for (int i = 0, j = possiblePath.Count - 1; i < possiblePath.Count; i++, j--)
24                  {
25                      if (Vector2.Distance(possiblePath[j].Position, ability[i]) > marginError) break;
26                      // FinalPath
27                      if (j == 0)
28                          return _abilityPaths[index].AbilityPathName;
29                  }
30              }
31          }
32      }
33  }
34  return null;
35 }
36 }
37 }
```

Figura 3.15: Comprobación de trazados en habilidades registradas

Una vez implementada la lógica de resolución de trazados, solo queda llamarla cuando se suelta el botón de ataque (clic izquierdo), momento en el que se comprobará y ejecutará la acción correspondiente encuentre o no la habilidad.

3.4.2.3 Ralentizar el tiempo

Dado que el objetivo es crear un combate rápido, puede surgir el problema de que el jugador se pueda poner nervioso y no piense con claridad qué habilidad usar. Para mitigar este problema, se planteó la posibilidad de hacer que el entorno se moviera más despacio durante un tiempo, permitiendo al jugador realizar los trazados con mayor calma.

Para implementar esto correctamente, se utiliza la variable *Time.unscaledDeltaTime* que proporciona Unity para aquellas lógicas o físicas que no deben verse afectadas por la escala del tiempo, que por defecto es 1. Esto es útil para que tanto contadores como correrías sigan funcionando perfectamente.

Con una sencilla línea de código se logra el efecto de ralentización modificando la escala del tiempo (*Time.timeScale*) a valores menores de 1, haciendo que todas las físicas y acciones que utilicen *Time.deltaTime* se vean afectadas.

Además, se ha implementado una correría que interpola estos dos valores, de modo que el cambio en la velocidad del juego no sea repentino, sino suave y gradual.

A continuación se explicará los diferentes tipos de habilidades que podemos trazar y encontrar a lo largo del juego y su funcionamiento más al detalle.

3.4.2.4 Imbuir

La habilidad de imbuir permite al jugador, mediante un trazado, dotar al personaje de un elemento específico. Esta habilidad es fundamental para poder realizar otros tipos de habilidades.

Si el jugador no utiliza esta primera habilidad, no podrá ejecutar ninguna otra.

Además, este efecto tiene una duración limitada, por lo que durante ese tiempo, si el jugador ataca a una entidad, se aplicará un daño adicional.

3.4.2.5 Pasiva

Como se mencionó anteriormente, es necesario estar en el estado "imbuido" para poder realizar cualquier otra habilidad.

Las habilidades pasivas son aquellas que directamente no actúan contra un enemigo o entidad dañable, sino que proporcionan beneficios en segundo plano al jugador.

Por ejemplo, una habilidad que ilumina una zona o aumenta la defensa del jugador.

3.4.2.6 Activa

Las habilidades activas son aquellas que deben ejecutarse contra una entidad dañable.

Para poder realizar estas habilidades, no solo es necesario estar en el estado "imbuido", sino que también se debe golpear al enemigo en las direcciones correspondientes a esta. Por ejemplo:

Imagina una habilidad con los trazados de la figura 3.14. Para marcar el nodo superior izquierdo, sería necesario hacer un trazado desde ese punto hasta el nodo inferior derecho ($\vec{V} = (0.707, -0.707)$). Repitiendo este proceso para los cuatro nodos que lo compone, podríamos realizar la habilidad.

Esta habilidad requiere estar dentro de un rango para ejecutarse, es decir, si se marca a un enemigo con los puntos para una habilidad, esta no funcionará a menos que estés a una distancia razonable.

Además, existe la posibilidad de combinar dos habilidades. En combate, se puede cambiar de personaje (explicado en 3.4.3.3), permitiendo que un enemigo tenga mas de un elemento marcado en un mismo nodo. Esto favorece a la mecánica con la posibilidad de realizar combinaciones de habilidades.

Es importante señalar que la forma de marcar estos nodos y cómo saber si se pueden marcar se explica con más detalle en el siguiente punto.

3.4.3 Sistema de combate

EL sistema de combate, como se ha mencionado anteriormente, se basa en el uso de trazados para reducir la vida de los enemigos.

Existen dos tipos de golpes, un golpe en una única dirección y las habilidades que conjuntan varias direcciones.

Para poder atacar en la dirección correcta, o al menos en la que el jugador desea, se ha implementado un sistema de fijado.

3.4.3.1 Fijado de enemigos

El sistema de fijado consta de ocho nodos que aparecerán frente a los enemigos (similar a la figura 3.10) y estarán orientados hacia la cámara. Esto ayuda a indicar la dirección en la que el jugador debe atacar y muestra qué nodos son vulnerables cuales están marcados.

Este sistema incluye un área delimitada que al accionar el botón de fijado, detecta a los enemigos dentro del rango y permite alternar entre ellos mientras se siga accionando. Si el jugador golpea a un enemigo sin fijarlo, este se marcará automáticamente y se desmarcará si sale del rango.

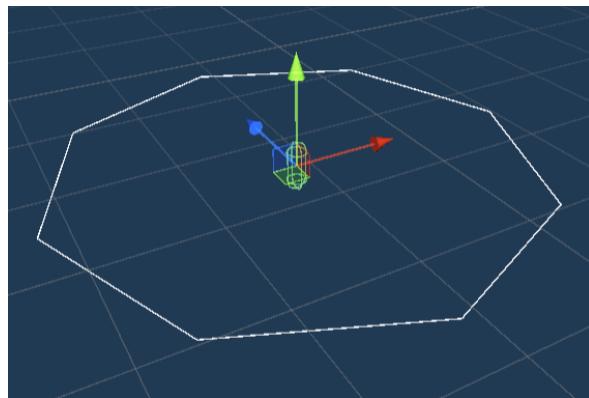


Figura 3.16: Debug visual del rango de fijado de enemigos

3.4.3.2 Bloqueos

El sistema de bloqueos es la principal mecánica defensiva de los enemigos.

Como se mencionó, este sistema consta de los 8 puntos que aparecen al fijar un enemigo y pueden cambiar entre dos estados.

- **Estado vulnerable:** En este estado, el enemigo puede recibir daño si se le golpea en ese nodo.
- **Estado bloqueado:** Si se ataca a un nodo bloqueado, el atacante quedará paralizado por un breve período de tiempo, quedando totalmente indefenso.

El cambio o asignación de estos estados se maneja en el componente **DefenseController**, el cual se encarga de gestionar los estados de los nodos y sus posibles marcas de elementos, devolviendo el estado de cada nodo en el sistema de combate.

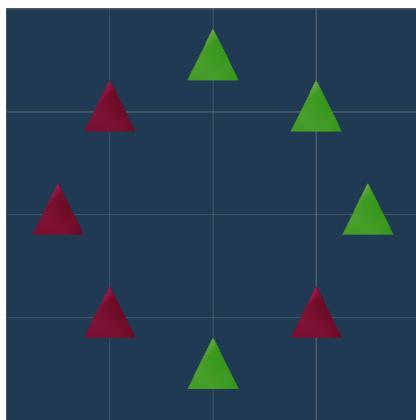


Figura 3.17: Estados bloqueos

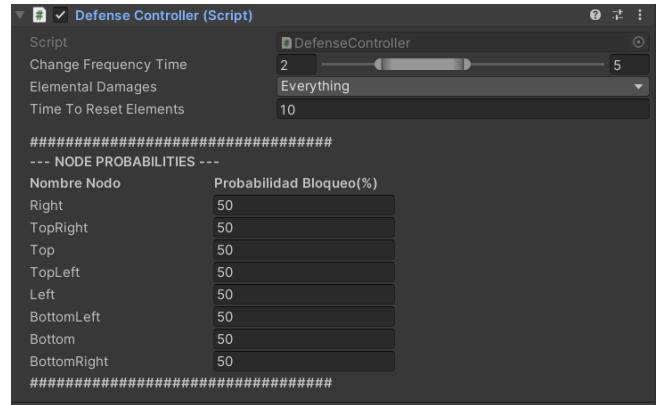


Figura 3.18: Inspector personalizado para **DefenseController**

Como se puede ver en la figura 3.18, para comportamientos más complejos, se puede registrar los nodos por los que el jugador suele atacar con mayor frecuencia y modificar la

probabilidad de bloqueo en tiempo de ejecución, añadiendo una capa adicional de desafío.

3.4.3.3 Cambio personaje

El juego cuenta con varios personajes, como se ha mencionado anteriormente, por lo que es necesario un sistema que permita cambiar de un personaje a otro de manera rápida y sin interrupciones significativas.

Este cambio debe poderse realizar mientras se está en el suelo o ejecutando alguna habilidad, limitando el cambio en otros momentos y añadiendo un tiempo de penalización entre cambios de los mismos personajes.

Para el funcionamiento de este sistema, se crea el componente **TeamController**, que maneja cuál es el personaje actual y almacena la información general de los personajes, así como su gestión.

El cambio se realiza mediante una animación en la que el personaje salta y aparece otro en la dirección opuesta de manera suave y fluida.

3.5 Inteligencia Artificial

La inteligencia artificial (IA) es un componente crucial para el proyecto. Por ello, se ha diseñado un sistema que permite crear un Behaviour Tree integrado con nodos que gestionan la Utility AI. Esta combinación permite tener dos tipos de estructuras en un mismo código: comportamientos secuenciales y rutinarios, así como acciones basadas en la utilidad.

3.5.1 Behaviour Tree + Utility AI

En esta sección se describe la integración del Behaviour Tree con el Utility AI para gestionar el comportamiento de las entidades de manera eficiente y versátil.

Además, se ha desarrollado un editor personalizado que facilita la creación de comportamientos y proporciona una estructura clara y visual.

3.5.1.1 Nodos

Como se ha explicado previamente en el **marco teórico**, se procederá a detallar los tipos de nodos que componen este sistema combinado de *Behaviour Tree + Utility AI*.

- **Nodos del Behaviour Tree**

- **Root:** Punto de inicio del árbol.
 - **Sequence:** Ejecuta sus hijos en orden secuencial hasta que uno falla.
 - **Selector:** Ejecuta sus hijos en orden hasta que uno tiene éxito.
 - **Priority Selector:** Similar al nodo Selector, pero ejecuta los nodos según un orden de prioridad.
-

- **Random Selector:** Selecciona y ejecuta un hijo de manera aleatoria.
 - **Loop:** Repite la ejecución de sus hijos hasta que un nodo dependiente (otro Behaviour Tree) falle.
 - **Dependency Sequence:** Similar al nodo Sequence, pero con una dependencia que puede hacer que el nodo falle aunque esté en estado *Running*.
 - **Inverter:** Invierte el resultado del estado de su nodo hijo.
 - **Leaf:** Nodo que ejecuta la acción específica.
- **Nodos de Utility AI**
 - **Action:** Nodo que ejecuta el comportamiento de la utilidad.
 - **Factor:** Nodo asociado a las acciones que obtiene el valor de la puntuación de un factor normalizado de 0 a 1 (por ejemplo, distancia al jugador, vida, etc.).
 - **Inverter:** Similar al del Behaviour Tree, pero invierte el valor del Factor.
 - **Bucket:** Nodo que almacena un conjunto de nodos Action y calcula la puntuación de sus hijos (media, sumatorio, etc.). Si este nodo tiene la puntuación más alta, ejecutará la acción con la mayor puntuación.
 - **Selection:** Nodo principal para la lógica de utilidad, cuyos hijos solo pueden ser Action o Bucket.

Una vez establecidos los nodos y su funcionamiento, cabe destacar que los nodos de *Utility AI* en este caso tienen un funcionamiento ligeramente distinto a lo explicado anteriormente.

En este sistema, los nodos de factores incluyen un nuevo parámetro: una gráfica que evalúa la puntuación obtenida del factor en un rango de 0 a 1. Añadir este parámetro mejora la inteligencia artificial.

Por ejemplo, al evaluar la puntuación de la vida en función de la gráfica (Figura 3.19), obtendríamos valores relativamente altos hasta la mitad de la vida. Sin embargo, cuando la vida baja de la mitad, la puntuación resultante disminuye drásticamente, haciendo que este factor sea menos relevante.

Esta capacidad de ajustar el valor de la utilidad permite que un personaje cambie su comportamiento dinámicamente a partir del mismo código base.

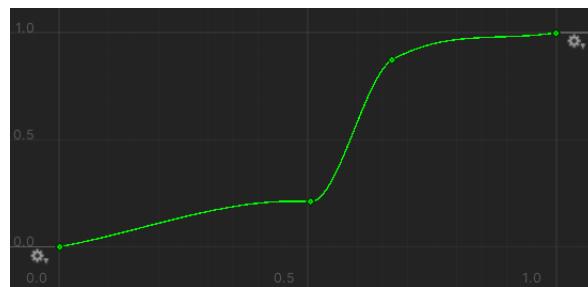


Figura 3.19: Gráfica factor vida

3.5.1.2 Editor personalizado

Para facilitar la creación de comportamientos sin necesidad de escribir código, se ha desarrollado un editor personalizado para cada tipo de nodo, junto con una ventana *Graph View* que permite programar el comportamiento en una interfaz independiente.

En esta interfaz, se asocia al nodo *Root* el *script* que contiene todas las funciones del comportamiento. Para los nodos que requieren una función específica, el editor muestra una lista con los métodos compatibles con el tipo de nodo, permitiendo que toda la creación de comportamientos se realice desde el editor.

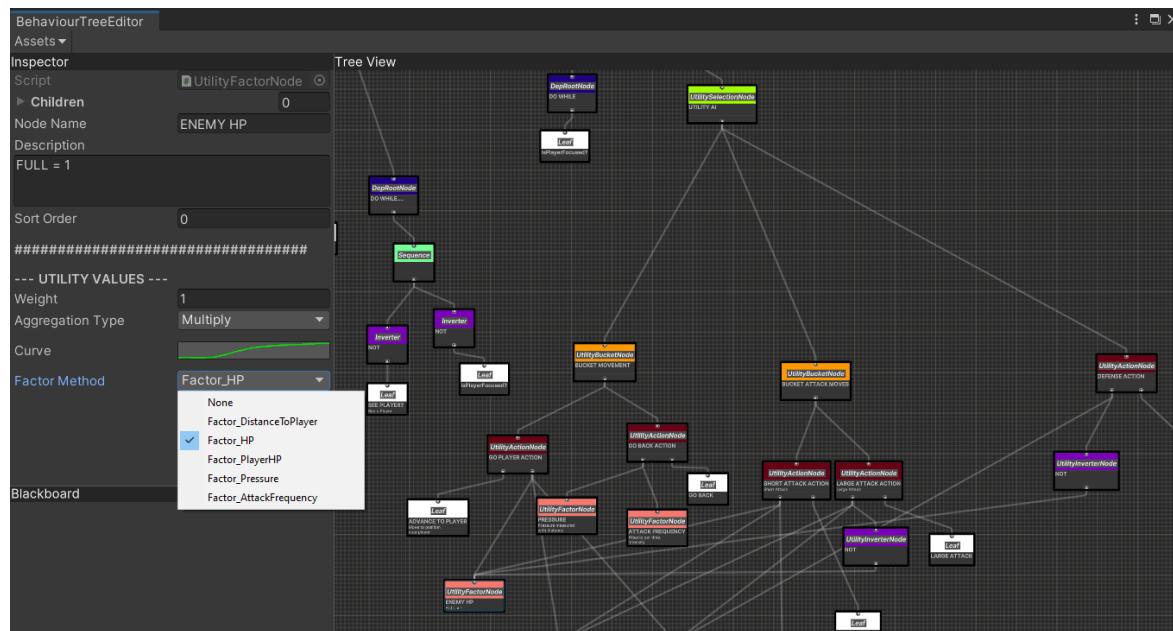


Figura 3.20: Ejemplo editor de nodos personalizado IA

3.6 Enemigos

Una vez implementados los sistemas que facilitan la creación de estructuras de comportamiento, se procederá a diseñar y desarrollar diferentes tipos de enemigos con estructuras simples. Esto permitirá mostrar y entender claramente el funcionamiento de los sistemas en conjunto.

Estos enemigos se basarán en una clase común llamada *AIBase*. Este componente garantizará que todos los enemigos tengan la misma estructura básica. Contendrá métodos para gestionar el movimiento del personaje a través de un *NavMesh*, el manejo del campo de visión, y una referencia al árbol de comportamiento que deben seguir.

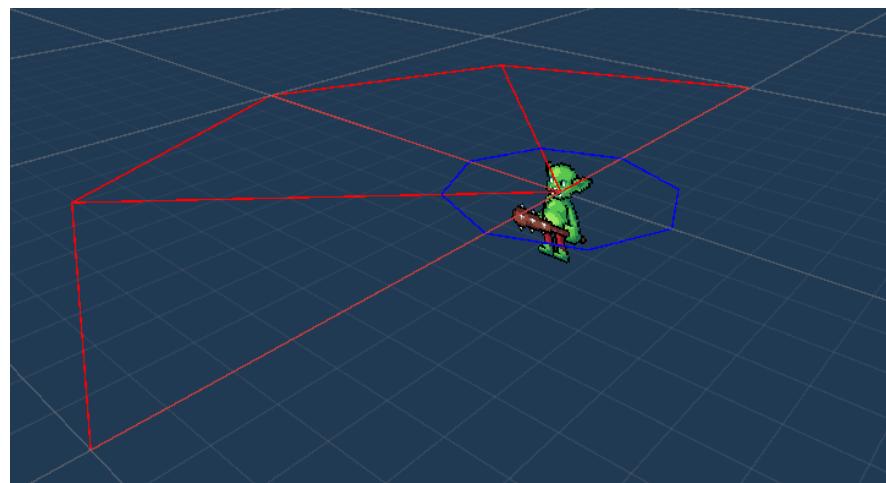


Figura 3.21: Campo de visión de los enemigos

Como se puede ver en la figura 3.21, los enemigos tienen un campo de visión personalizable, permitiendo modificar el ángulo y el rango para crear diferentes tipos de enemigos con distintas capacidades de visión.

Para este proyecto, se han diseñado tres tipos de enemigos: uno de corto alcance, uno de largo alcance, y finalmente un jefe.

- Goblin



Figura 3.22: Sprite del Goblin (*autor: Mattz Art*)

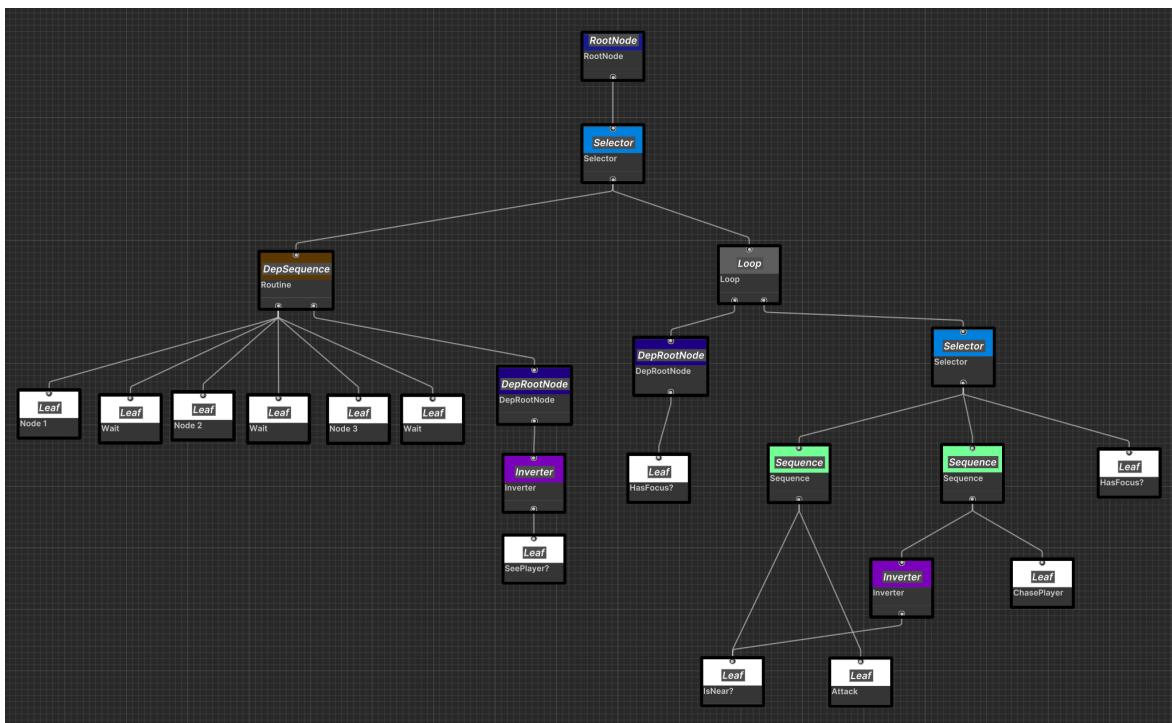


Figura 3.23: Behaviour Tree Goblin

La figura 3.23 muestra la estructura del árbol de comportamiento diseñado para el Goblin. Este enemigo tiene un comportamiento agresivo de corto alcance. Posee dos fases: una en la que patrulla una zona y otra en la que, al detectar al jugador, lo persigue y ataca hasta perderlo de vista.

- Demon



Figura 3.24: Sprite de Demon (*autor: Mattz Art*)

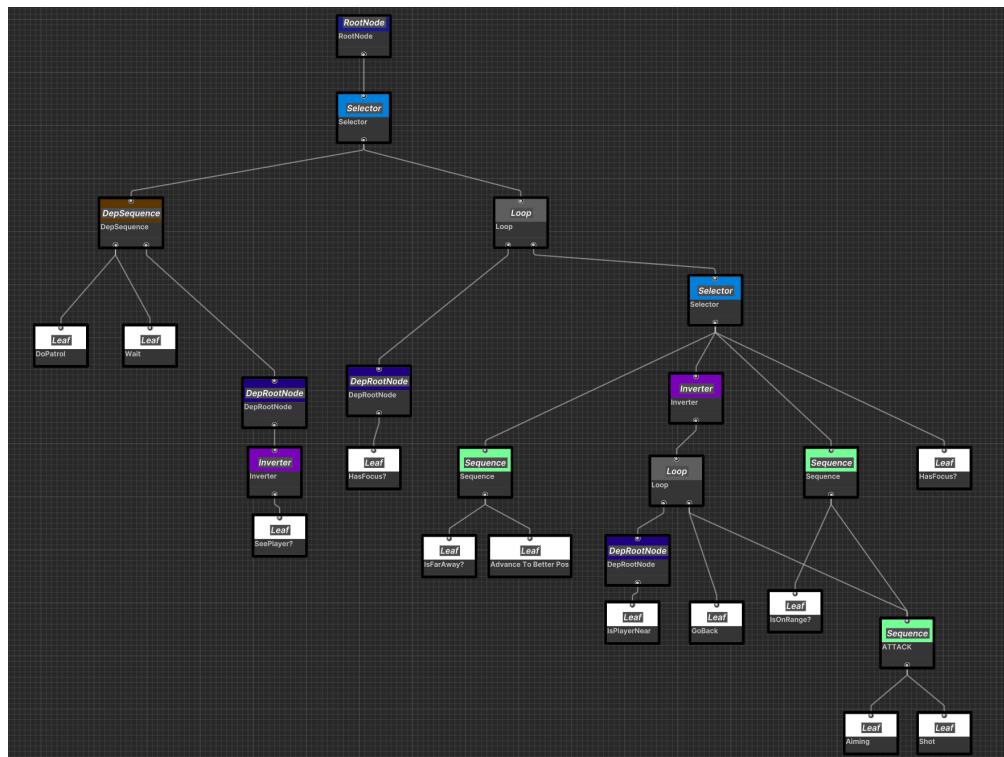


Figura 3.25: Behaviour Tree Demon

La figura 3.25 muestra el comportamiento de un enemigo contrario al Goblin. Este demonio mantiene la distancia y dispara proyectiles. Patrulla hasta que detecta al jugador, momento en el cual se acerca hasta una distancia cómoda para él. Si el jugador se acerca o se aleja, el demonio ajusta su posición para mantener su zona de confort.

En la siguiente figura se aprecia visualmente esta zona, representada por un círculo naranja y amarillo.

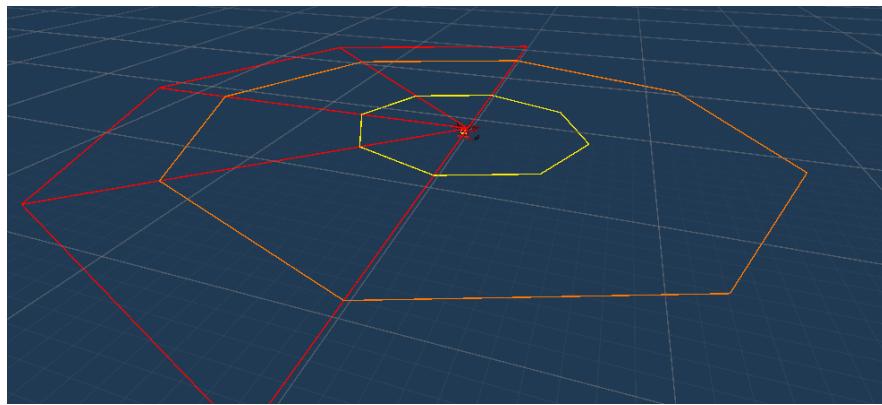


Figura 3.26: Zona confort Demon

- **Boss Hand**



Figura 3.27: Sprite de Boss Hand (*autor: Gauna*)

Este enemigo tiene un comportamiento más complejo que los anteriores, demostrando el potencial del sistema de inteligencia artificial al utilizar nodos del *Utility AI*, lo que le otorga una personalidad más dinámica.

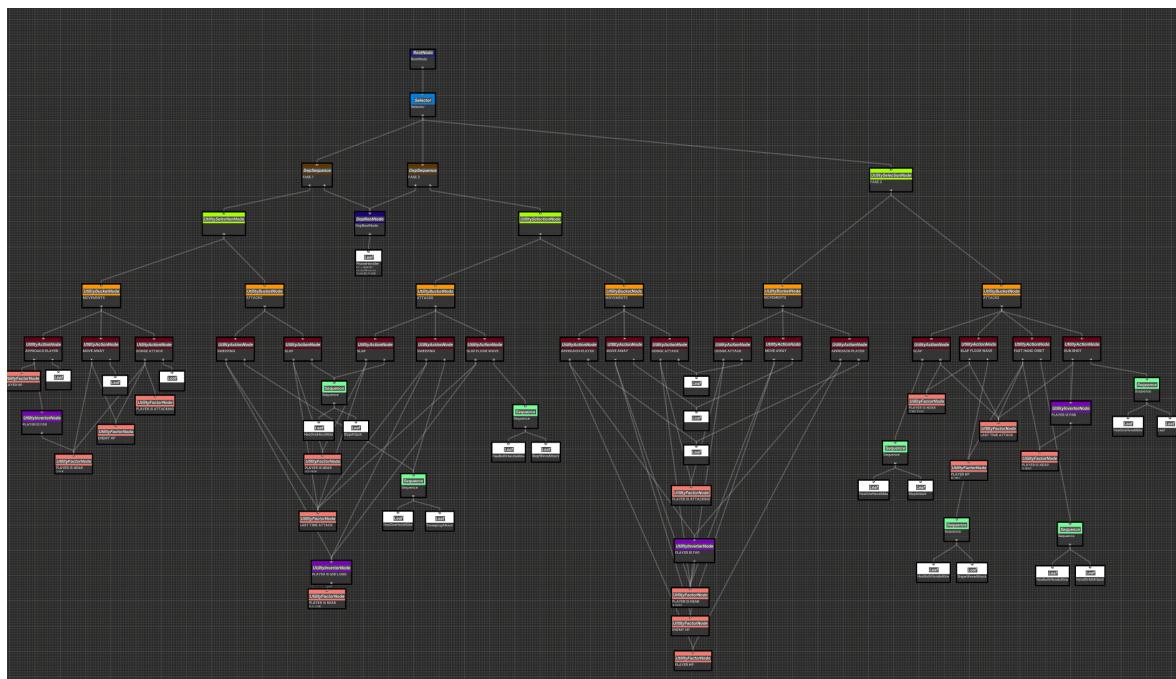


Figura 3.28: Behaviour Tree Boss Hand

En la figura 3.28, muestra el comportamiento completo del jefe. Este enemigo es mucho más complejo y se divide en tres fases. La diferencia en cada fase radica en la selección de sus ataques, los cuales son similares pero incluyen una predicción del ataque para que el jugador tenga la oportunidad de esquivarlo.



Figura 3.29: Predicción de los ataques de Boss Hand

Para realizar esta predicción se ha utilizado el componente **DecalProjector** proporcionado por Unity.

Al igual que el jugador puede evitar los ataques del jefe, este puede esquivar los del jugador realizando un *dash* a un punto estratégico. Este desplazamiento se realiza mediante coordenadas polares, tomando como origen la posición del jugador y moviéndose al punto más cercano y estratégico para evitar el ataque.

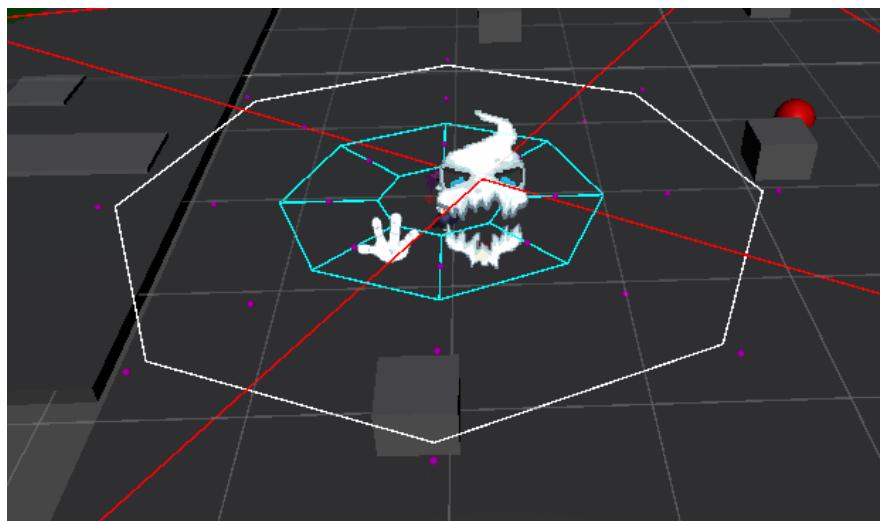


Figura 3.30: Desplazamiento con coordenadas polares

En cuanto al movimiento y los demás ataques, los comportamientos son similares a los mencionados para otros enemigos.

La utilización de los nodos de utilidad en la Figura 3.28 permite que el enemigo realice comprobaciones y cálculos para considerar su salud, distancia al jugador y otros factores para la ejecución de sus acciones.

Para la realización de cada ataque, se comprueba si hay el número de manos disponible en caso de ser un ataque que requiera ambas o alguna en específico.

3.7 Personajes principales

En este proyecto es necesario crear diferentes personajes con sus pequeñas variaciones en habilidades o estadísticas.

Para una primera versión del proyecto y para probar el funcionamiento básico, se ha diseñado y desarrollado dos personajes que muestran todas las mecánicas principales ya mencionadas.

- Naro



Figura 3.31: Sprite de Naro (*autor: Mattz Art*)

Naro es el protagonista y primer personaje jugable. Su elemento base es el fuego y dispone de tres habilidades, una de cada tipo: imbuir, pasiva y activa.

- **Imbuir:** Naro se envuelve en un aura de fuego, permitiéndole golpear con este elemento para marcar o herir a sus enemigos.
- **Antorcha:** Habilidad pasiva que invoca una llama, proporcionando luz durante un tiempo para explorar zonas oscuras.
- **Tornado de fuego:** Habilidad activa que invoca un tornado de fuego sobre el enemigo, causando un gran daño.

- Anciano



Figura 3.32: Sprite del anciano (*autor: Mattz Art*)

El Anciano es el acompañante del protagonista y se une al equipo en un momento del juego. Su elemento base es la tierra y dispone de tres habilidades.

- **Imbuir:** Habilidad básica que envuelve al personaje en un aura marrón, otorgando mayor resistencia a los golpes y permitiendo usar otras habilidades de tierra.
- **Muro de tierra:** Habilidad pasiva que invoca un muro de tierra, útil para bloquear ataques enemigos o pasar por ciertas áreas.

3.8 Sistema de diálogos

Para mostrar diálogos o información relevante en la pantalla es necesario un sistema de diálogos eficiente.

El desafío radica en contar la historia de manera fluida, es decir, sin limitaciones ni repeticiones en los diálogos. Por ejemplo, hablar con un NPC y que pueda haber múltiples diálogos diferentes. Para ello, se necesita un sistema que permita crear ramificaciones y variaciones en los diálogos, haciendo la historia más dinámica y envolvente.

Primeramente, se encontró un código desarrollado por otro programador el cual concedió el permiso para utilizarlo y modificarlo en este proyecto (enlace al código y autor).

Aunque este código era cómodo y simple de usar, tenía sus limitaciones y no podía manejar eventos para que cambiase los diálogos de una forma más amplia. Es por eso que se diseñó y desarrolló un sistema propio desde cero que completara todas las funciones necesarias.

La idea era crear un sistema de diálogos intuitivo que pudiera ser manejado de manera visual en una ventana de Unity, tomando como referencia el sistema empleado en el editor personalizado de la IA.

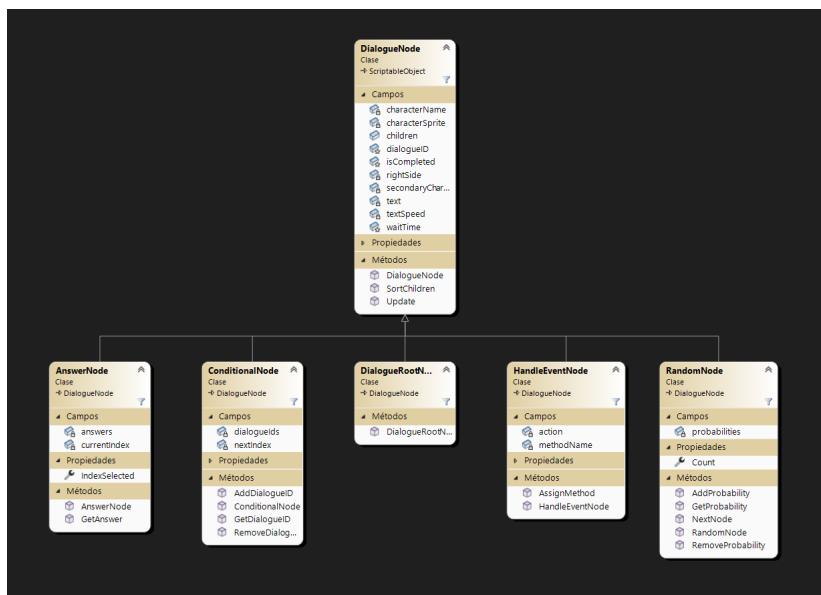


Figura 3.33: Diagrama de clase Sistema Diálogos

3.8.1 Nodos y Dialogue Manager

En la figura (Figura 3.33) se puede ver la estructura planteada y desarrollada del sistema. El diálogo está estructurado en forma de árbol el cual se recorrerá en una única dirección. Los nodos que componen este árbol son los siguientes:

- **Dialogue:** Nodo base que almacena toda la información necesaria para un mensaje de diálogo, como el sprite, texto, velocidad, ID, entre otros parámetros.
- **Answer:** Nodo de respuesta que muestra una lista de hasta tres respuestas para que el jugador elija una. Cada respuesta puede llevar a otros diferentes nodos.
- **Conditional:** Nodo que comprueba si se ha cumplido una condición para continuar la ejecución en ambos resultados. Las condiciones pueden ser de varios tipos ya que se gestionan mediante un ID en los diálogos o uno personalizado. Estas condiciones las gestionará el **DialogueManager**, quién tendrá todos los IDs obtenidos durante el juego.
- **DialogueRoot:** Nodo raíz desde donde empieza todo diálogo.
- **HandleEvent:** Nodo que ejecuta un evento o función almacenada en el mismo *frame*, permitiendo realizar animaciones, efectos u otras acciones independientes del sistema de diálogo.
- **Random:** Nodo que selecciona aleatoriamente un hijo en función de una probabilidad establecida para cada uno. De esta forma se puede hacer diálogos variantes para un NPC.

Funcionamiento del sistema

Para crear un diálogo se utiliza el componente **DialogueComponent**, que tiene una referencia a un *ScriptableObject* que contiene los datos de un diálogo y una colisión. En sus parámetros se especifica cómo se activa este diálogo, ya sea por colisión, al iniciar una escena o simplemente llamando a su método *StartDialogue()*.

Una vez se activa, se envía un evento al **DialogueManager**, que establece el estado de diálogo, habilitando así los controles del mismo y desactivando los normales. El **DialogueManager** muestra el texto con sus respectivas animaciones y maneja la lógica para avanzar entre nodos cuando el jugador pulsa un botón.

3.8.2 Editor personalizado

Al igual que en el editor personalizado de la IA, se ha desarrollado un editor siguiendo el mismo sistema de *GraphView*, adaptando la estética para adecuarse al sistema de diálogos y permitir una visualización clara y sencilla de los nodos.

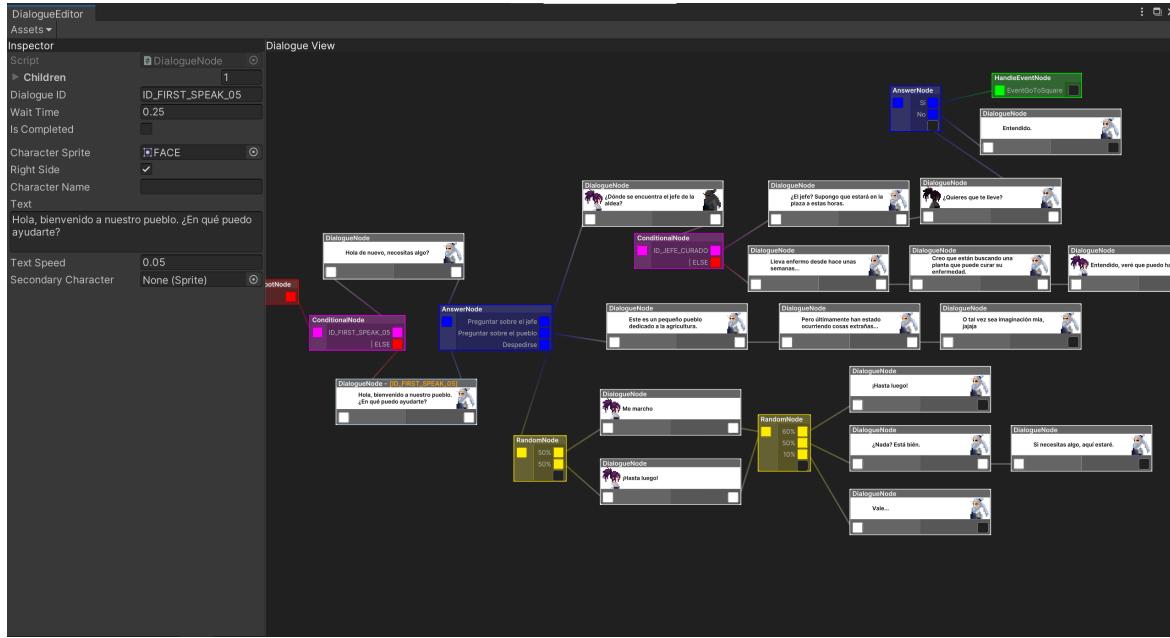


Figura 3.34: Ejemplo editor editor de nodos personalizado diálogos

En la figura (3.34), podemos ver todos los nodos y su funcionamiento de una manera clara e intuitiva. En el árbol, se puede ver que dependiendo de varios factores, la conversación puede seguir diferentes ramas. Por ejemplo, si el jugador vuelve a hablar con el mismo NPC, la primera frase del diálogo cambia porque el NPC ya reconoce al jugador.

3.9 Sistema de cinemáticas

Las cinemáticas en los videojuegos son secuencias de vídeo o animación utilizadas para narrar una historia, desarrollar personajes o crear momentos de impacto emocional. Estas secuencias pueden variar desde simples diálogos entre personajes, hasta escenas complejas con efectos visuales y movimientos complejos.

Para este proyecto, es esencial desarrollar un sistema que permita realizar animaciones con movimientos de cámara, diálogos durante un evento, o simplemente cambios de plano.

Existen diversas formas de crear una cinemática, como pre-renderizando la secuencia o simplemente utilizando el motor del juego para ejecutarla en tiempo real. Dado que este es un proyecto simple y gráficamente no muy exigente, se ha decidido que las cinemáticas se ejecuten en tiempo real.

Para gestionar las cinemáticas, se utiliza el sistema de animaciones de Unity, *Timeline*, como base. Para poder ejecutar una secuencia, se necesita el componente *PlayableDirector*. Sin embargo, para este proyecto se busca un sistema capaz de complementarse con el sistema de diálogo.

Para lograr esto, se ha creado un componente llamado **CinematicDirector**, el cual requiere un **PlayableDirector** para su funcionamiento. Este objeto es el que almacenará la lógica de una cinemática, es decir, el *Timeline* con las acciones, los métodos necesarios para la creación de diálogos y la realización de pausas o reanudaciones en función de los eventos que ocurran.

Este enfoque permite pausar y reanudar animaciones en función de un diálogo, consiguiendo ejecutar ciertas acciones de una cinemática cuando se cumple una condición. Para lograrlo, se hace uso del Sistema de Eventos basado en el patrón *Observer*, y se aprovechan las **Signals** del *Timeline*, eventos que ejecutan una función de un script en un tiempo determinado.

Una cinemática puede empezar de diferentes maneras: mediante un *trigger* que la ejecute, al cargar la escena, o referenciando el objeto desde otro *script* para controlar el momento de su ejecución.

Para gestionar cuando debe cargarse una cinemática, se ha creado un objeto llamado **CinematicManager**. Este se encarga de comprobar para cada escena qué cinemáticas han sido ejecutadas y cuáles no, cargando así solo las necesarias. Esto optimiza el uso de recursos y organiza mejor el flujo de cinemáticas.

Además, considerando que el juego abarca múltiples situaciones, similar al sistema de diálogos, se ha desarrollado una funcionalidad para manejar dependencias entre cinemáticas. Cuando se carga la escena, el gestor verifica si la cinemática a cargar depende de otra, permitiendo una gestión ordenada y clara de múltiples cinemáticas en un mismo nivel.

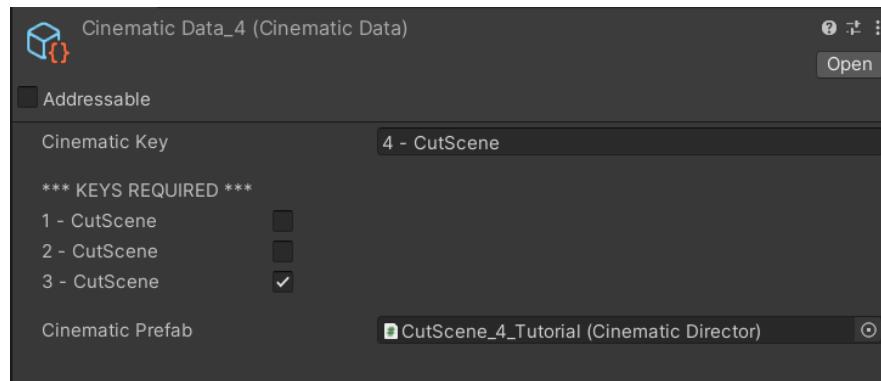


Figura 3.35: ScriptableObject de una cinemática

3.10 Interfaz de Usuario (UI)

La interfaz de usuario es un punto importante, no solo para la estética, sino que también proporciona información vital que puede influir en la jugabilidad. Por ejemplo, permite al jugador ver cuánta vida le queda, identificar qué enemigo está fijado para atacar, entre otras funcionalidades importantes.

Para este proyecto, se ha diseñado tanto la interfaz de los menús, así como la que aparece durante la partida. Sin embargo, debido a los objetivos de este proyecto, algunos de los assets se han adquirido de terceros con sus respectivas licencias y consentimientos a través de *Itch.io* y la *Unity Asset Store*.

3.10.1 Diseño menús

En esta sección se presenta el diseño de los menús del juego, incluyendo el menú principal, el menú de pausa y el menú de *Game Over*. Estos elementos de la interfaz de usuario son fundamentales para la navegación y la experiencia del jugador. A continuación, se muestran los bocetos de cada menú.

- Menú Principal

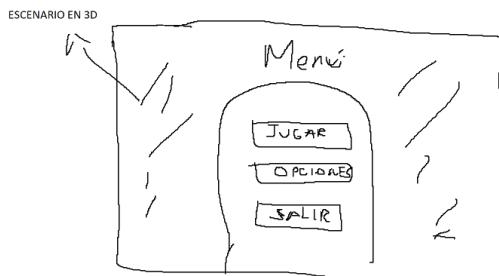


Figura 3.36: Boceto y diseño del menú principal

La idea principal es desarrollar un menú con un fondo 3D mínimamente animado para añadir dinamismo, junto con animaciones en cada botón para mejorar la interacción.

- Menú Pausa

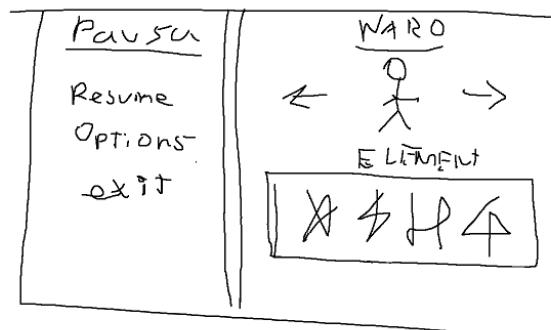


Figura 3.37: Boceto y diseño del menú de pausa

El menú de pausa es sencillo y muestra el personaje que el jugador está controlando, su elemento base y una lista de los trazados que puede realizar, todo acompañado de animaciones.

- Menú Game Over



Figura 3.38: Boceto y diseño del menú Game Over

El menú de fin de partida presenta una animación del personaje cayendo al suelo, con dos opciones: repetir el nivel o regresar al menú principal.

La gestión de todos estos menús está vinculada al **StateManager** del menú, que se encarga de habilitar y deshabilitar los *inputs* necesarios, así como de gestionar las transiciones entre los diferentes estados del menú.

3.10.2 Diseño HUD In-Game

En esta sección se presenta el diseño del HUD (*Heads-Up Display*) *In-Game*, el cual incluye elementos cruciales para la experiencia del jugador. Estos elementos son los que superpone a cualquier otro en la escena y sirve a modo de información, normalmente relevante, para el

jugador sin interrumpir la jugabilidad.

Se detallará el boceto del HUD principal, diálogos y cuadro de avisos. Estos componentes están diseñados para mantener al jugador informado, mejorando la interacción y la inmersión en el juego.

- **HUD Principal**



Figura 3.39: Boceto y diseño del HUD principal

El HUD principal se encargará de mostrar información vital para el jugador, incluyendo la salud restante, el tiempo restante del elemento imbuido y el medidor de concentración, que facilita la ralentización del tiempo para el uso de habilidades.

- **Diálogos**

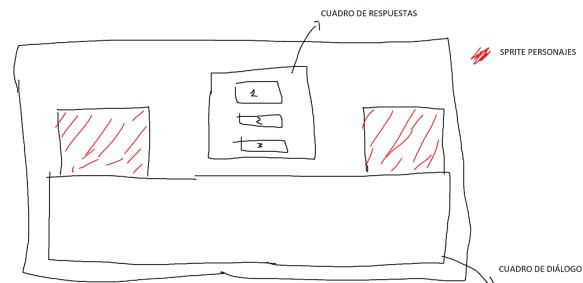


Figura 3.40: Boceto y diseño del HUD diálogos

La estructura planteada para los diálogos sigue el estilo típico de los videojuegos *RPG*. Los personajes que están hablando se muestran a los laterales, con un cuadro de diálogo en la parte inferior. En el centro aparece una ventana con las posibles respuestas a una conversación, siguiendo el esquema desarrollado en secciones anteriores.

- Aviso

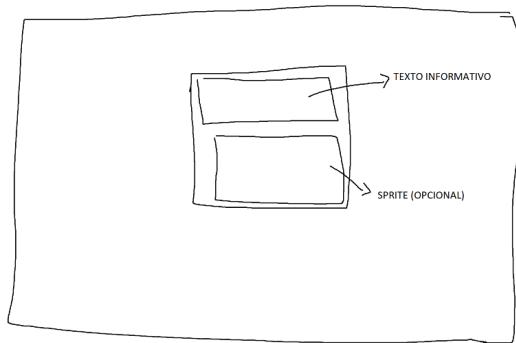


Figura 3.41: Boceto y diseño del HUD avisos

Este elemento se utiliza para mostrar información importante durante el juego, como instrucciones del tutorial, la obtención de un objeto, o cuando un personaje deja el grupo y ya no es jugable. Consta de un texto con la información relevante y, opcionalmente, un sprite para representar un personaje u otro tipo de elemento visual.

Para gestionar el funcionamiento de estos elementos en el juego, se utiliza frecuentemente el `Event System` previamente mencionado. Además, se ha creado un *script* llamado `HUDManager` el cual se encarga de determinar cuándo debe mostrarse cada componente del HUD. Este *script* utiliza animaciones para asegurar que la presentación de los elementos sea visualmente atractiva y clara.

3.11 Flujo del juego

En esta sección se describe el flujo del juego, detallando cómo los diferentes elementos interactúan y se integran para proporcionar una experiencia de usuario cohesiva y fluida.

Se presentará un diagrama que ilustra el recorrido del juego, desde su inicio hasta las posibles situaciones de finalización.

Asimismo, se describirá el diseño del nivel, sus distintas zonas y la progresión dentro de un estilo *metroidvania*, donde es importante adquirir habilidades para acceder a nuevas áreas.

3.11.1 Diagrama de flujo

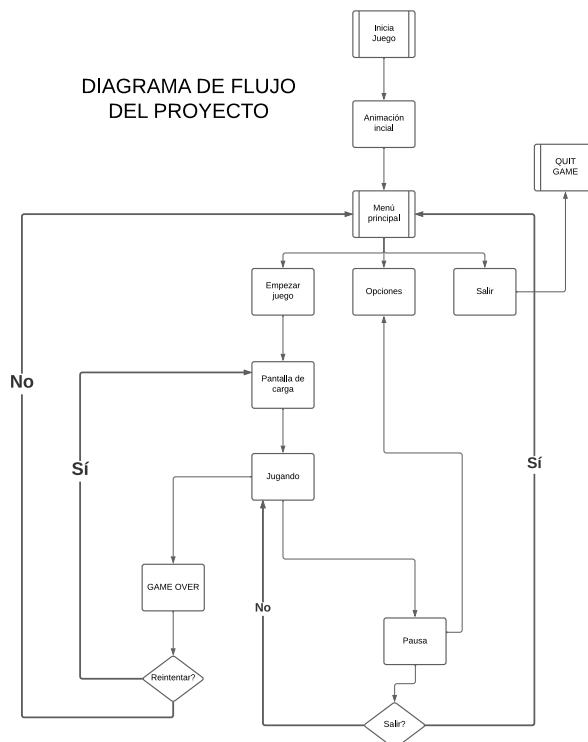


Figura 3.42: Diagrama de flujo del proyecto

3.11.2 Nivel, zonas y progresión

El nivel consta de una única escena la cual se utiliza todos los aspectos desarrollados del juego además de un *script* adicional para adaptar la cámara a cada zona de la escena y facilitar la jugabilidad.

Este *script* es llamado **CameraZone** y consta de una colisión y una cámara. cuando el jugador entra en la zona, se realiza una transición de una cámara a la otra de manera suave y

sin que el jugador pierda su orientación.

Una vez se consigue una forma de desplazarse por un escenario con una cámara inteligente, se diseña y desarrolla el nivel que contendrá todas las acciones y elementos desarrollados.

El jugador aparecerá en una zona inicial junto a una cinemática que muestre el inicio y la introducción al juego. Una vez se finaliza esta acción el jugador tendrá libertad total de moverse por el escenario siguiendo diferentes caminos.

Los caminos finalizarán en zonas con diferentes habilidades como recompensa los cuales para su acceso, es necesario haber desbloqueado una previamente.

3.12 Post-procesado y efectos

El post-procesado es una técnica que aplica filtros y efectos a la imagen de la cámara antes de que se muestre en pantalla. Esta técnica puede mejorar significativamente la calidad visual de un videojuego con una configuración mínima (Documentation, 2017).

Utilizando el post-procesado, es posible emular las propiedades físicas de las cámaras y películas tradicionales, implementando efectos como *Bloom*, *Depth of Field*, *Chromatic Aberration* y *Color Grading*. Estos efectos no solo mejoran la calidad visual del juego, sino que también aumentan la inmersión del jugador.

Junto a estos efectos visuales, las animaciones de movimiento juegan un papel crucial en la creación de una experiencia de juego fluida y dinámica. En este proyecto, se emplea el uso del *asset DOTween*, una potente biblioteca de animación para Unity, que simplifica la creación de transiciones y animaciones suaves. *DOTween* permite animar propiedades como la posición, rotación y escala de los objetos de manera eficaz y con un mínimo de código. Este se usa para menús o animaciones sencillas del juego.

3.12.1 Shaders

Los *shaders* son un aspecto importante en la creación del apartado gráfico de un videojuego. Estos programas permiten a los desarrolladores crear efectos visuales variados y espectaculares, mejorando significativamente la calidad y la inmersión en el juego. A través de estos, se pueden lograr efectos de iluminación que van desde lo realista hasta lo estilizado, añadiendo profundidad y atmósfera a las escenas (Fernández, 2023).

Además, los *shaders* permiten simular materiales reflectantes y refractivos, como espejos, gafas y cristales, generando efectos visuales impresionantes y realistas. Otro uso destacado es en la creación de efectos de agua, permitiendo representar desde superficies tranquilas hasta olas en movimiento y cascadas realistas.

En este proyecto, se han desarrollado varios *shaders* para demostrar su potencial y una versión más acabada en el apartado gráfico. Entre ellos se encuentra una cascada que se ha necesitado para que complementara con el escenario realizado a partir de assets de la *Unity*

Asset Store. Además, se ha desarrollado un *shader* que añade un contorno (*outline*) a los *sprites*, junto con un efecto de aura de poder para mostrar a los personajes de manera diferente al tener un elemento imbuido.

Todos estos han sido desarrollados mediante el *ShaderGraph* de Unity.

- **Cascada:**

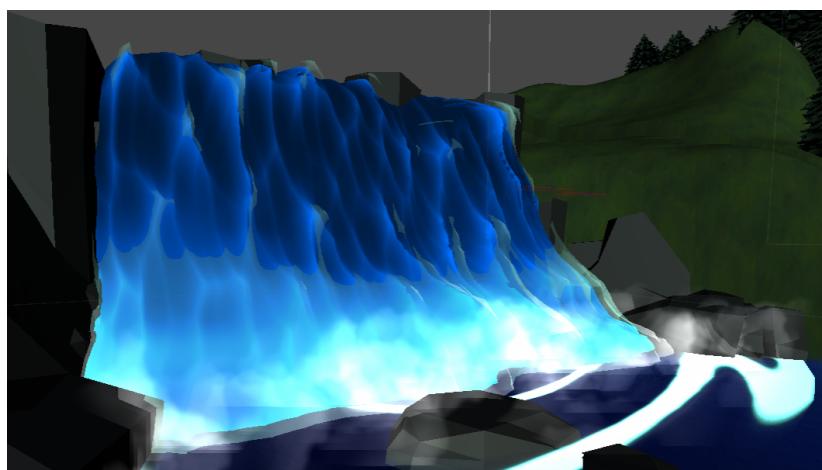


Figura 3.43: Efecto cascada con *Unity ShaderGraph*

Para la creación de este *shader*, se ha seguido un tutorial, añadiendo un efecto personalizado con partículas de humo para hacer la cascada más densa a la hora de colisionar con el río y crear una coherencia con el estilo del escenario (Prod., 2019).

- **Estado imbuido:**

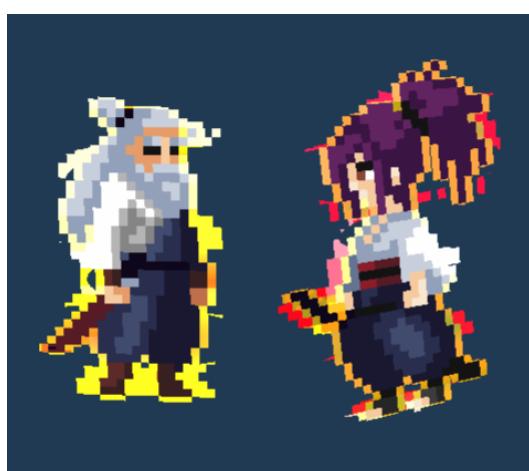


Figura 3.44: Efecto estado imbuido *Unity ShaderGraph*

SpriteOutlineAnimationShader	
Shader Graphs	
MainTex	Texture2D
Thickness	Float
OutlineColor	Color
PowerThickness	Float
PowerSpeed	Float
PowerColor	Color
VoronoiDensity	Float
VoronoiAngle	Float
PowerTiling	Vector2

Figura 3.45: Propiedades del *shader* imbuido

Este *shader* se ha desarrollado a partir de lo aprendido en varios tutoriales, incluyendo el de la cascada. En este, se realiza un desplazamiento del sprite (*Texture2D*) para generar un borde y luego colorearlo según el elemento, utilizando el nodo *Tiling and Offset* junto con operaciones de suma y resta para lograr el efecto deseado.

En la siguiente figura se muestra un segmento del *ShaderGraph*, donde se aplica el mismo algoritmo para las cuatro direcciones: arriba, abajo, y los laterales. (Figura 3.46).

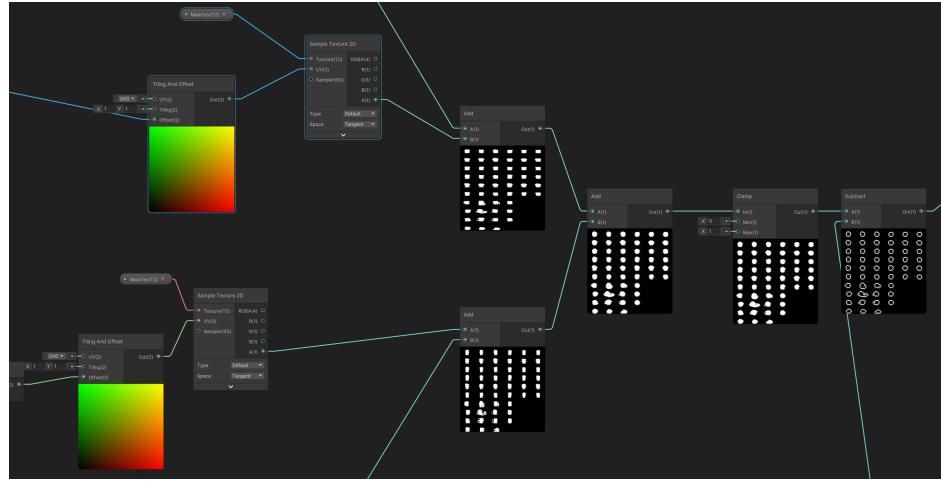


Figura 3.46: *ShaderGraph outline* de sprites

Además, mediante un filtro *Voronoi* combinado con el bucle del juego, se crea un efecto de aura alrededor del personaje, añadiendo un toque vivo y llamativo al efecto del borde.

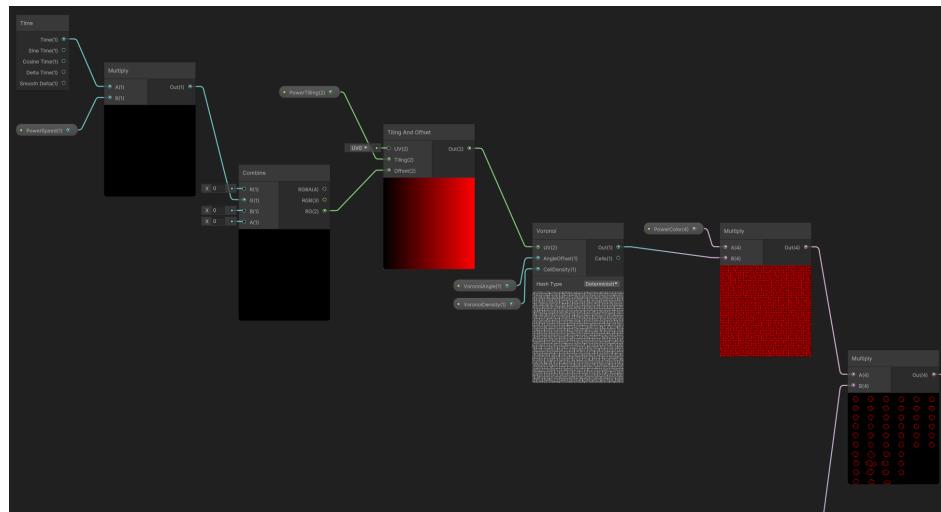


Figura 3.47: *ShaderGraph* del efecto imbuido

3.12.2 *Vignette* y *Bloom*

Estos efectos de post-procesado se añaden gracias a la utilización del *Universal Render Pipeline* (URP) en Unity. El URP permite una modificación más detallada del motor, facilitando la adición de efectos de post-procesado como *Vignette* y *Bloom* en este caso.

Para aplicar estos efectos, es necesario ajustar la configuración del proyecto y crear un *GameObject* genérico que contenga el componente *Volume*, permitiendo añadir varios efectos a la cámara (Figura 3.50).

- ***Vignette*:** Este efecto se utiliza para cambiar el ambiente del juego durante la ejecución de una habilidad. Al ralentizar el tiempo, se puede indicar visualmente este cambio aplicando un efecto de viñeta, creando así un ambiente de concentración.



Figura 3.48: Efecto *Vignette*

- ***Bloom*:** Este efecto hace que los elementos del juego emitan una luz utilizando una técnica conocida como *Glow*. Al habilitar el uso de color HDR en los *shaders*, se introduce una variable de intensidad que se combina con los parámetros de *Bloom*, creando este efecto. En el proyecto, esta técnica se utiliza para los estados de imbuido (Figura 3.44) o para efectos que requieran que el material tenga un efecto de emisión.

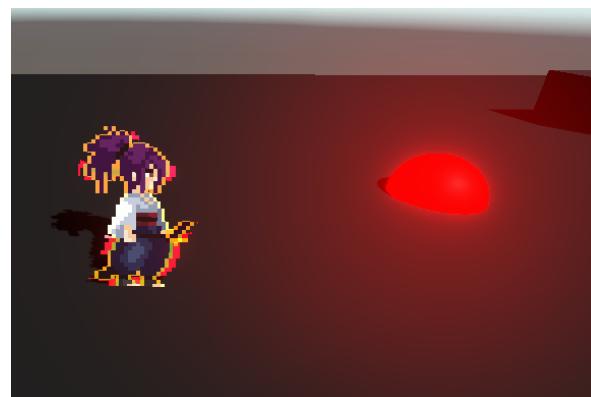


Figura 3.49: Efecto *Glow*

Para entender mejor en qué consiste este efecto, en la figura (Figura 3.49) se puede observar cómo la esfera de la derecha emite un rojo muy intenso en comparación con el efecto de imbuir. Esta diferencia depende del valor de la intensidad del color.

Para gestionar y modificar estos parámetros, se ha creado un *script* llamado **PostProcessingManager** que se encarga de su administración.

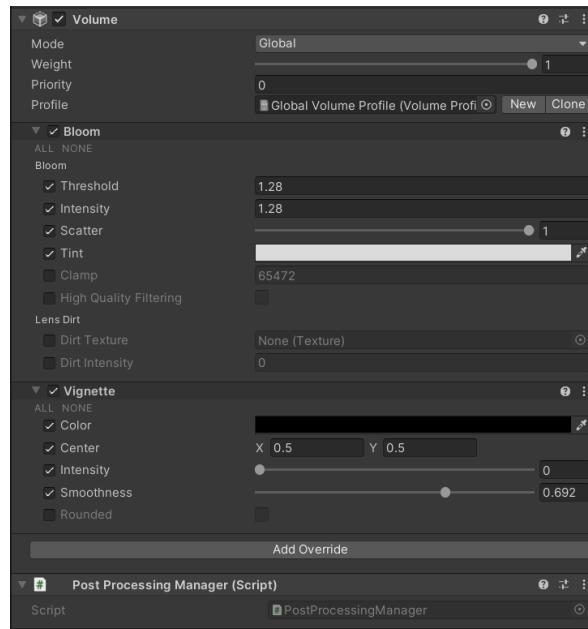


Figura 3.50: *Post Processing Manager*

4 Resultados

En este capítulo se presentan los resultados finales del proyecto, ilustrados mediante una serie de imágenes y descripciones detalladas. Los resultados se organizan siguiendo un caso de uso, mostrando el desarrollo final de los menús y áreas del juego conforme al flujo planteado en la figura 3.11.1.

Al iniciar el proyecto, se muestra el menú principal.



Figura 4.1: Menú principal (Logo: *AI Logo Maker*)

Al seleccionar la opción de iniciar partida, aparece una pantalla de carga seguida de una primera cinemática con un pequeño diálogo.

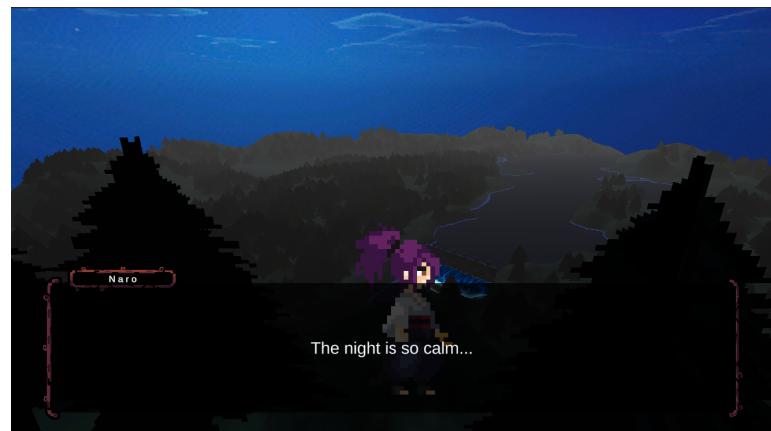


Figura 4.2: Cinemática inicial

Al finalizar esta cinemática, se carga otra escena donde aparece el jugador y una ventana que explica los diferentes controles habilitados.



Figura 4.3: Inicio del nivel con controles

Avanzando en el nivel, se inicia una cinemática y un diálogo con un personaje que explica el entorno del juego y proporciona un arma al jugador.



Figura 4.4: Cinemática con diálogo

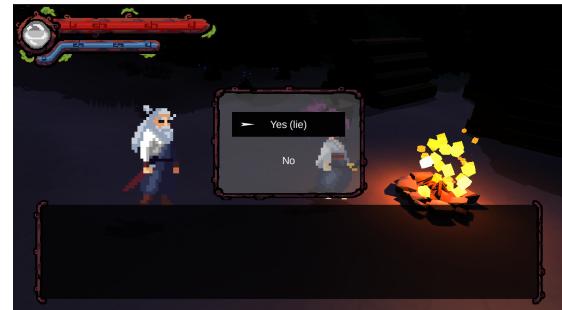


Figura 4.5: Opciones de respuesta en diálogos

A continuación, aparecen varias ventanas, como las de la figura 4.3, mostrando cómo fijar y ejecutar ataques simples en una única dirección.

Avanzando por uno de los caminos, se llega a una zona sin salida con dos enemigos.



Figura 4.6: Fijado de enemigos

Al herir a un enemigo o al jugador, se realiza un efecto de destello para indicar daño.



Figura 4.7: Efecto de daño en el jugador

En el caso de los enemigos, además del destello, se liberan partículas y se muestra la dirección del trazado.



Figura 4.8: Golpe a enemigo

Cerca de esta zona, existe una área oculta con varios enemigos y una pared escalable una vez se obtiene la habilidad necesaria. El enemigo Demon intentará disparar y quitar vida al jugador.



Figura 4.9: Zona escalada

Siguiendo el camino anterior, se llega a una zona donde se debe saltar sin caer al agua.



Figura 4.10: Zona de plataformas



Figura 4.11: Zona de plataformas 2

Al avanzar, se llega a la zona final donde aparece una cinemática. Un anciano explica cómo realizar habilidades y permite imbuir un elemento.



Figura 4.12: Obtención de la primera habilidad

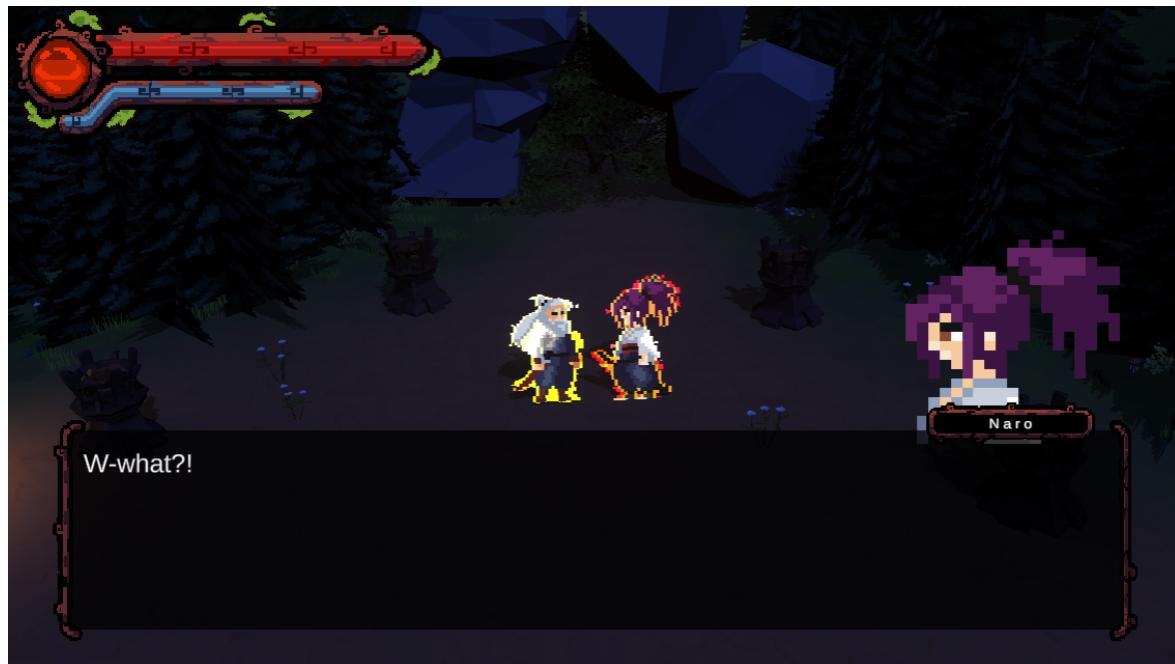


Figura 4.13: Estado de imbuir

Con esta habilidad, al golpear antorchas, estas se encenderán, permitiendo acceder a una nueva zona opcional.



Figura 4.14: Mini puzzle para abrir una puerta

En esta nueva zona, se encuentra un cofre con la habilidad de hacer *dash* y más adelante, unas setas con físicas de rebote que requieren dicha habilidad para avanzar.



Figura 4.15: Setas con rebote

Al final del camino se encuentra la habilidad de escalar paredes, permitiendo volver al camino inicial.



Figura 4.16: Obtención de la habilidad de salto entre paredes

Una vez conseguido todo esto de manera opcional, se puede volver al inicio donde se vio el salto de la pared y conseguir un ataque de fuego como habilidad activa.

Al final del nivel y del camino principal, se encuentra una aldea donde se une el anciano, disponiendo ahora de dos personajes jugables y enfrentando un jefe que ataca la aldea.



Figura 4.17: Unión de un nuevo personaje

Al abrir el menú de pausa, se muestran datos como las habilidades del personaje actual y su elemento.



Figura 4.18: Menú de pausa



Figura 4.19: Habilidad activa (Fuego)



Figura 4.20: Habilidad pasiva (Tierra)



Figura 4.21: Combate contra un jefe

Esta es la última zona demostrable de la *Vertical Slice*. En caso de ser derrotados, aparece el menú *Game Over*, con la posibilidad de reintentar o salir al menú principal.

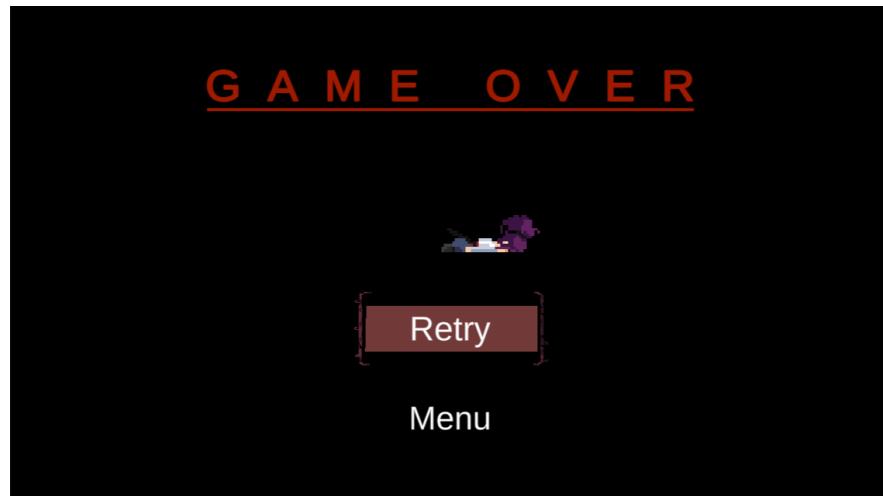


Figura 4.22: Estado *Game Over*

5 Conclusiones

En esta sección se abordarán las experiencias obtenidas durante la realización de este proyecto, destacando los aprendizajes más significativos y los desafíos enfrentados. Además, se discutirán las limitaciones encontradas a lo largo del desarrollo y se propondrán posibles mejoras para futuros trabajos similares.

5.1 Aprendizaje y experiencia en el desarrollo

Durante el desarrollo de este proyecto, se han adquirido valiosos conocimientos en diversas áreas, así como una mejora significativa en la capacidad de resolución de problemas.

La implementación de todos los sistemas en conjunto, siguiendo patrones de diseño que facilitan la escalabilidad, ha sido uno de los aspectos más valiosos en términos de experiencia personal. Estos patrones permiten una lectura y comprensión más clara del código, lo cual es crucial para el mantenimiento y evolución de cualquier proyecto.

La programación de las diferentes mecánicas, las físicas o el sistema de trazado, me ha ayudado a poder aplicar los conocimientos matemáticos aprendidos durante estos años de carrera obteniendo un resultado esperado y claro.

Si tuviera que destacar un aspecto en particular, sería la creación de los editores para los sistemas de inteligencia artificial y diálogos. Este desafío fue considerable debido a mi limitado conocimiento sobre *GraphView*, *UIBuilder* o Editor de Unity. Sin embargo, tras superar estos obstáculos, se logró un resultado final muy satisfactorio en ambos editores y sistemas.

Finalmente, haber creado una *Vertical Slice* me ha permitido tener una visión completa de todos los aspectos involucrados en el desarrollo de un videojuego. Esta experiencia me ha ayudado a decidir en qué área enfocar mis futuros esfuerzos.

5.2 Limitaciones y posibles mejoras

Al considerar este proyecto como una *Vertical Slice*, se han identificado varias limitaciones y posibles mejoras durante su desarrollo.

Un ejemplo significativo es el sistema de trazado. Una posible mejora sería la implementación de un giroscopio como entrada, utilizando dispositivos como el mando de la Nintendo Switch o cualquier otro que contenga esta tecnología. Esto podría ser un punto a favor, ya que permitiría añadir una capa adicional de diversión para aquellos jugadores que dispongan

de esta tecnología.

Además, debido a que el trabajo se ha centrado principalmente en el apartado técnico, apenas ha habido tiempo para dedicar al aspecto artístico. Como resultado, algunos elementos visuales carecen de cohesión debido a la diversidad de assets adquiridos durante el desarrollo. Sin embargo, gracias al código desarrollado, el proyecto puede centrarse en mejorar el diseño de niveles y el apartado artístico en el futuro. Esto se debe a que muchas cosas pueden cambiarse sin necesidad de modificar el código, debido a la consistencia en mantener un código limpio y escalable.

Otra área con potencial para mejoras es la inteligencia artificial de los enemigos. Actualmente, los enemigos tienen comportamientos básicos, pero se podría añadir más vida y dinamismo a sus acciones, haciéndolos más complejos y desafiantes. Implementar estos comportamientos enriquecería la experiencia de juego.

Finalmente, se podrían desarrollar más habilidades para los personajes, aumentando así la variedad de acciones disponibles. La introducción de estas, no solo enriquecería la jugabilidad, sino que también permitiría diseñar niveles mucho más atractivos, ofreciendo a los jugadores una experiencia más completa.

Bibliografía

- Anusky. (2020). *Ori and the blind forest y la magia que lo envuelve*. Descargado de <https://gamingfrommars.com/articulos/ori-and-the-blind-forest-y-la-magia-que-lo-envuelve>
- de Byl, P. (2022). *Learn advanced ai for games with behaviour trees*. Descargado de https://www.udemy.com/share/104R603@Ud40I4VGHftUDTL-IsSVZTaDdJLotavPwgj_0-3RxL0EH0S4r6PfYoCtDLB75WoqXQ==/
- Documentation, U. (2017). *Visión general del post-procesamiento*. Descargado de [https://docs.unity3d.com/es/2017.4/Manual/PostProcessingOverview.html#:~:text=Post-processing%20\(post-procesamiento,se%20muestre%20en%20la%20pantalla](https://docs.unity3d.com/es/2017.4/Manual/PostProcessingOverview.html#:~:text=Post-processing%20(post-procesamiento,se%20muestre%20en%20la%20pantalla).
- Fernández, E. C. (2023). *Descubriendo los shaders de unity*. Descargado de <https://www.tokioschool.com/noticias/shaders-unity/>
- Foqum. (s.f.). *Finite state machine*. Descargado de <https://foqum.io/blog/termino/finite-state-machine/>
- García, D. E. (2019). *Ventajas y diferencias entre unity, unreal engine y godot*. Descargado de <https://openwebinars.net/blog/ventajas-diferencias-unity-unreal-engine-godot/>
- Graham, D. . (2014). An introduction to utility theory. En S. Rabin (Ed.), *Game ai pro: Collected wisdom of game ai professionals* (pp. 113–126). A K Peters/CRC Press.
- Knight, S. (2021). *What are 2.5d games? how they differ from 2d and 3d games*. Descargado de <https://www.makeuseof.com/what-are-2-5d-games-2d-3d/>
- Król, L. W. (s.f.). *Finite state machines*. Descargado de <https://www.universityofgames.net/articles/finite-state-machines-in-unity/>
- Marco. (2016). *Finite state machine for game developers*. Descargado de <https://gamedevelopertips.com/finite-state-machine-game-developers/>
- MasterClass. (2021). Metroidvania games: 5 characteristics of metroidvania games. *MasterClass*. Descargado de <https://www.masterclass.com/articles/metrodvania-definition>
- Mira, A. R. (2020). *La inteligencia artificial en videojuegos*. Descargado de <https://www.tokioschool.com/noticias/inteligencia-artificial-videojuegos/>
- Parra, D. (2023). *Patrones de diseño para videojuegos*. Descargado de <https://www.udemy.com/share/108MQW/>

Prod., G. A. (2019). *Unity shader graph - waterfall effect tutorial*. Descargado de <https://youtu.be/yJ0NRr-DdYU>

Sabiq. (2023). *Unity vs unreal vs godot - comparison, pros, cons*. Descargado de <https://imeta.tech/blog/unity-unreal-godot-comparison>

TheKiwiCoder. (2021). *Unity / create behaviour trees using ui builder, graphview, and scriptable objects [ai #11]*. Descargado de <https://youtu.be/nKpM98I7PeM>

TheShaggyDev. (2023). *An introduction to utility ai*. Descargado de <https://shaggydev.com/2023/04/19/utility-ai/>

Wikipedia contributors. (2024a). *2.5d — Wikipedia, the free encyclopedia*. Descargado de <https://en.wikipedia.org/w/index.php?title=2.5D&oldid=1221435676>

Wikipedia contributors. (2024b). *Metroidvania — Wikipedia, the free encyclopedia*. Descargado de <https://en.wikipedia.org/w/index.php?title=Metroidvania&oldid=1228936685>
