PONTIFICIA UNIVERSIDAD CATÓLICA DEL ECUADOR

FACULTAD DE INGENIERÍA

ESCUELA DE SISTEMAS

TRABAJO DE TITULACIÓN

DESARROLLO DE UN PROTOTIPO DE VIDEOJUEGO EN 2D TIPO PLATAFORMAS.

AUTOR

HECTOR DANIEL IBARRA MONTOYA

TUTOR

ARCOS VILLAGÓMEZ SUYANA FABIOLA

JESUITAS ECUADOR

Tabla de Contenidos

TEMA *8*

PLANTEAMIENTO DEL PROBLEMA *9*

JUSTIFICACIÓN *10*

OBJETIVOS: GENERAL Y ESPECÍFICOS **11**

Objetivo General 11

Objetivos Específicos 11

MARCO TEÓRICO Y CONCEPTUAL **12**

Marco Teórico 12

Marco Conceptual 12

ALCANCE **14**

FUNDAMENTACION TEÓRICA *15*

Revisión de Godot Engine y sus características **15**

Sistema de nodos y escenas 15

Lenguajes de programación soportados 15

Herramientas integradas 15

Exportación multiplataforma 15

Pequeña vista a la ventana de GODOT 4 16

Justificación de uso de GODOT 4 para este proyecto 17

Aplicación de metodologías ágiles en el desarrollo de software **19**

Justificación del uso de Scrum 19

Adaptación de Scrum para un solo desarrollador (Scrum Guide, 2025) 19

Estructuras de datos y algoritmos en videojuegos **21**

Estructuras de datos y su aplicación en videojuegos 21

Algoritmos clásicos en videojuegos 22

Comportamientos de movimiento 22

Pool de objetos 23
Principios de interacción humano-computador 24
UI y UX como elementos narrativos 24
Consistencia en la interfaz de usuario 24
Uso de colores en la UI 24
Reducción de latencia en la respuesta del sistema 24
Uso de metáforas en la UI 25
Inteligencia artificial aplicada a videojuegos 26
Comportamiento de enemigos 26
Movimiento de personajes 26
Aprendizaje y mejora 26
DESARROLLO DEL TRABAJO 28
Planificación del proyecto utilizando Scrum 28
Roles en Scrum 28
Creación del Product Backlog 28
Planificación de Sprints 29
Daily Scrum 29
Revisión y Retrospectiva del Sprint 29
Herramientas Utilizadas 29
Diseño Conceptual y Técnico del Videojuego 30
Diseño Conceptual 30
Diseño Técnico 32
Implementación del prototipo en Godot Engine 34
Integración de recursos gráficos y sonoros 64
Análisis de Resultados 72
Pruebas internas y validación del prototipo 72
Evaluación interna de usabilidad y experiencia de usuario 75

Funcionalidades de la UI 75

Experiencia de Control y Movilidad 75

Análisis del cumplimiento de los objetivos planteados 77

Estructuras de Datos y Algoritmos 77

Ingeniería de Software 79

Interacción Humano-Computador 79

Inteligencia Artificial 82

CONCLUSIONES Y RECOMENDACIONES 85

Conclusiones del proyecto 85

Recomendaciones para futuras investigaciones 87

ANEXO 1 88

ANEXO 2 90

SPRINT BACKLOG 90

DAILY SCRUM 94

RETROSPECTIVA DE SPRINTS 102

ANEXO 3 114

ANEXO 4 116

ALGORITMO DEL JUGADOR 116

Algoritmo de jugador (ENZO) 116

ALGORITMOS DE ENEMIGOS 120

Algoritmo IA BÁSICA Ranas 120

Algoritmo IA BÁSICA noback dragón 122

Algoritmo IA MEDIA Samuráis 3, 4 y 6 125

Algoritmo IA Alta Old Samurái 128

ALGORTIMO IA SUPERIOR Elf Ranger 131

ALGORITMOS DE OBJETOS 137

Algoritmo Baúl 137

Algoritmo Coins 139

Algoritmo Shuriken 141

ANEXO GRÁFICO 143

BIBLIOGRAFÍA 147

Tabla de Figuras

Figura 1. Ventana principal de Godot 16
Figura 2. Vista de Script en el editor de Godot 17
Figura 3. Vista 2D 17
Figura 4. Ventana tras abrir Godot 34
Figura 5. Vista inicial del proyecto en Godot 35
Figura 6. Vista alejada de nivel 36
Figura 7. Inicio de nivel 37
Figura 8. Algunos enemigos en escena 37
Figura 9. Samurai_6 en escena 38
Figura 10. Zona de combate final 38
Figura 11. Menú inicial 39
Figura 12. Menú de pausa 39
Figura 13. Menú de opciones 40
Figura 14. Menú de controles 40
Figura 15. Tabla de puntuación 41
Figura 16. HUD Elementos que siempre se muestran en pantalla 42
Figura 17. Como se ve en el juego 42
Figura 18. Shuriken 43
Figura 19. Coins 43
Figura 20. Directorio 45
Figura 21. Nodos de la escena principal 46
Figura 22. Personaje Principal - Enzo 47
Figura 23. Acciones del Personaje 48
Figura 24. Nodos de Enzo 49

Figura 25. Dragon_noback 50
Figura 26. Fog_greenblue 50
Figura 27. Fog_purpleblue 51
Figura 28. Fog_blueblue 51
Figura 29. Estructura general de ranas 52
Figura 30. Estructura de dragón_noback 52
Figura 31. Samurai_6 53
Figura 32. Samurai_4 54
Figura 33. Samurai_3 54
Figura 34. Samurai_6 55
Figura 35. Samurai_4 55
Figura 36. Samurai_3 55
Figura 37. Old_Samurai 56
Figura 38. Nodos de Old_Samurai 57
Figura 39. Area2D de Samurai_3 58
Figura 40. Señal body_entered 58
Figura 41. Ejemplo de Q_table 60
Figura 42. Enemigo Principal - Elf Ranger 61
Figura 43. Nodos de elf_ranger 63
Figura 44. Q_table de elf_ranger 63
Figura 45. Añadiendo un nuevo nodo a la escena (ctrl + a) 64
Figura 46. Nodos AnimationPlayer, Sprite2D y CollisionShape2D 64
Figura 47. Imagen recién cargada en el editor 65
Figura 48. Animation en BottomPanel 66
Figura 49. Imagen cortada y animada 67

Figura 50. Nueva escena a partir de un nodo tipo “AudioStreamPlayer2D”......................68

Figura 51. Escena creada con el nodo principal para Audio...68

Figura 52. Pista de audio haunted 69

Figura 53. Import con haunted seleccionado 69

Figura 54. Inspector del nodo “AutioStreamPlayer2d” 70

Figura 55. Nodo AudioStreamPlayer2D agregado a la escena 70

Figura 56. Propiedades de "AudioStreamPlayer2D" 71

Figura 57. Vista de Baúl en tiempo real...79

Figura 58. Enzo en los primeros ciclos de desarrollo 143

Figura 59. Primer escenario construido 143

Figura 60. Primer escenario construido (2)...144

Figura 61. Enemigo de los primeros ciclos del desarrollo 144

Figura 62. Enemigo de los primeros ciclos del desarrollo (2) 145

Figura 63. Enemigos considerados en los ciclos iniciales 145

Figura 64. Enemigos considerados en los ciclos iniciales (2)..146

TEMA

Desarrollo de un prototipo de videojuego en 2D.

PLANTEAMIENTO DEL PROBLEMA

En el ámbito académico y profesional, existe una brecha entre los conocimientos teóricos adquiridos en las materias de informática e ingeniería de software y su aplicación práctica en proyectos complejos como el desarrollo de videojuegos. A pesar de que la universidad no ofrece una materia específica sobre desarrollo de videojuegos, los estudiantes cuentan con las habilidades y conocimientos necesarios para emprender tal proyecto. El problema principal radica en cómo integrar eficazmente estos conocimientos para desarrollar un prototipo funcional de un videojuego en 2D tipo plataformas utilizando Godot Engine.

Los problemas secundarios incluyen la gestión eficiente del proyecto mediante la metodología Scrum, la implementación de mecánicas de juego que incorporen estructuras de datos y algoritmos, el diseño de una interfaz de usuario intuitiva siguiendo los principios de interacción humano-computador, y la incorporación de elementos de inteligencia artificial para mejorar la experiencia de juego.

JUSTIFICACIÓN

El desarrollo de videojuegos es una industria en constante crecimiento y evolución tecnológica. La creación de un prototipo de videojuego en 2D tipo plataformas utilizando Godot Engine, una herramienta de código abierto ofrece la oportunidad de aplicar y consolidar conocimientos adquiridos en áreas fundamentales de la informática y la ingeniería de software. Este proyecto permitirá explorar aspectos técnicos como programación, algoritmos y pseudocódigos, estructuras de datos, inteligencia artificial, ingeniería de Software e interacción humano computador, enfocándose en la implementación de mecánicas de juego.

En base a los conocimientos adquiridos durante esta formación universitaria, como son Estructura de Datos, Ingeniería de Software, Interacción Humano Computador e Inteligencia Artificial, es posible desarrollar un prototipo que incorpore los principales elementos técnicos de un videojuego. Aunque las áreas artísticas no son el foco principal, es viable utilizar recursos disponibles en línea que son libres y accesibles para cualquiera. Este proyecto demostrará que, a pesar de que la universidad no ofrece una materia específica sobre desarrollo de videojuegos, sí proporciona la experiencia y el conocimiento necesarios para llevar a cabo uno.

OBJETIVOS: GENERAL Y ESPECÍFICOS

Objetivo General

Desarrollar un prototipo de videojuego en 2D tipo plataformas utilizando Godot Engine, aplicando la metodología Scrum e integrando conocimientos de Estructura de Datos, Algoritmos y Pseudocódigos, Ingeniería de Software, Interacción Humano-Computador e Inteligencia Artificial.

Objetivos Específicos

1. Introducción: Contextualizar el proyecto en el ámbito del desarrollo de videojuegos y justificar la elección de las tecnologías y metodologías utilizadas.
2. Fundamentación Teórica: Investigar y analizar los conceptos teóricos relacionados con el desarrollo de videojuegos, Godot Engine, metodologías ágiles y las áreas de conocimiento involucradas.
3. Desarrollo del Trabajo: Implementar el prototipo del videojuego aplicando las técnicas y conocimientos adquiridos, siguiendo las prácticas de ingeniería de software y Scrum.
4. Análisis de Resultados: Evaluar el prototipo desarrollado en términos de funcionalidad, eficiencia, usabilidad y cumplimiento de los objetivos planteados.
5. Conclusiones y Recomendaciones: Reflexionar sobre los resultados obtenidos, identificando logros, desafíos y proponiendo mejoras futuras.

MARCO TEÓRICO Y CONCEPTUAL

Antecedentes o Marco Referencial

El desarrollo de videojuegos independientes ha crecido significativamente gracias a herramientas accesibles y de código abierto como Godot Engine, que permiten a desarrolladores individuales crear juegos de calidad sin grandes recursos. La adopción de metodologías ágiles como Scrum ha mejorado la eficiencia y adaptabilidad en proyectos de software, lo que es aplicable al desarrollo de videojuegos.

En la carrera de Ingeniería en Sistemas de Información, no se ofrece una materia donde se aprenda a desarrollar videojuegos. Sin embargo, la formación proporcionada brinda las bases necesarias para abordar proyectos en este campo. Este proyecto aplica de manera práctica estos conocimientos para crear un prototipo de videojuego 2D, utilizando recursos disponibles en línea y enfocándose en la implementación técnica.

Marco Teórico

El proyecto se fundamenta en las siguientes áreas teóricas clave:

- Estructuras de Datos y Algoritmos: Aplicación eficiente para gestionar recursos y mecánicas del juego.
- Ingeniería de Software: Uso de la metodología ágil Scrum para la gestión del proyecto.
- Interacción Humano-Computador (IHC): Diseño de interfaces y experiencias centradas en el jugador para lograr una interacción intuitiva y guiada en los puntos necesarios.
- Inteligencia Artificial: Implementación de comportamientos autónomos en personajes no jugables para mejorar la jugabilidad.

Marco Conceptual

- Godot Engine: Motor de desarrollo de videojuegos de código abierto que soporta tanto 2D como 3D.
- Pixel Art: Estilo gráfico que utiliza imágenes de baja resolución para crear una estética retro.
- Metodología Scrum: Marco de trabajo ágil para la gestión y desarrollo de proyectos.

- Inteligencia Artificial en Videojuegos: Técnicas y algoritmos utilizados para dotar de comportamiento inteligente a los personajes del juego.

ALCANCE

El proyecto se centrará en el desarrollo de un prototipo funcional de un videojuego en 2D tipo plataformas para PC, que incluirá un único nivel, mecánicas de juego fundamentales, enemigos con comportamientos simples y medianamente complejos de inteligencia artificial y una interfaz de usuario diseñada bajo principios de usabilidad. No se abordará la comercialización del juego ni el desarrollo de niveles avanzados o modos multijugador.

FUNDAMENTACION TEÓRICA

Revisión de Godot Engine y sus características

Godot Engine es un motor de videojuegos de código abierto y multiplataforma que permite la creación de juegos en 2D y 3D. Su creciente popularidad se debe a su flexibilidad, facilidad de uso y modelo de licenciamiento permisivo (MIT), que permite a los desarrolladores utilizarlo sin restricciones comerciales (https://godotengine.org/license/, 2025). A continuación, se describen algunas de sus características más destacadas:

Sistema de nodos y escenas

Godot emplea una arquitectura basada en **nodos y escenas**, lo que facilita la organización jerárquica de los elementos del juego y fomenta la reutilización de componentes. Cada escena está compuesta por nodos que pueden representar objetos gráficos, sonidos, scripts o cualquier otro elemento funcional del juego. Este enfoque modular simplifica el desarrollo, permite una estructura más limpia y mejora la mantenibilidad del código.

Lenguajes de programación soportados

Godot ofrece soporte para múltiples lenguajes de programación, siendo **GDScript** su lenguaje nativo, diseñado específicamente para el desarrollo de videojuegos y con una sintaxis similar a Python. Además, admite **C#** y **C++**, lo que proporciona flexibilidad para optimizar el rendimiento en áreas críticas del juego.

Herramientas integradas

El motor cuenta con un editor visual intuitivo, un potente sistema de animaciones, un depurador integrado y soporte para físicas avanzadas. Su motor gráfico admite iluminación en tiempo real, sombreado avanzado y renderizado tanto para 2D como para 3D. Adicionalmente, Godot permite la personalización del editor mediante scripts, lo que lo convierte en una herramienta adaptable a distintas necesidades de desarrollo.

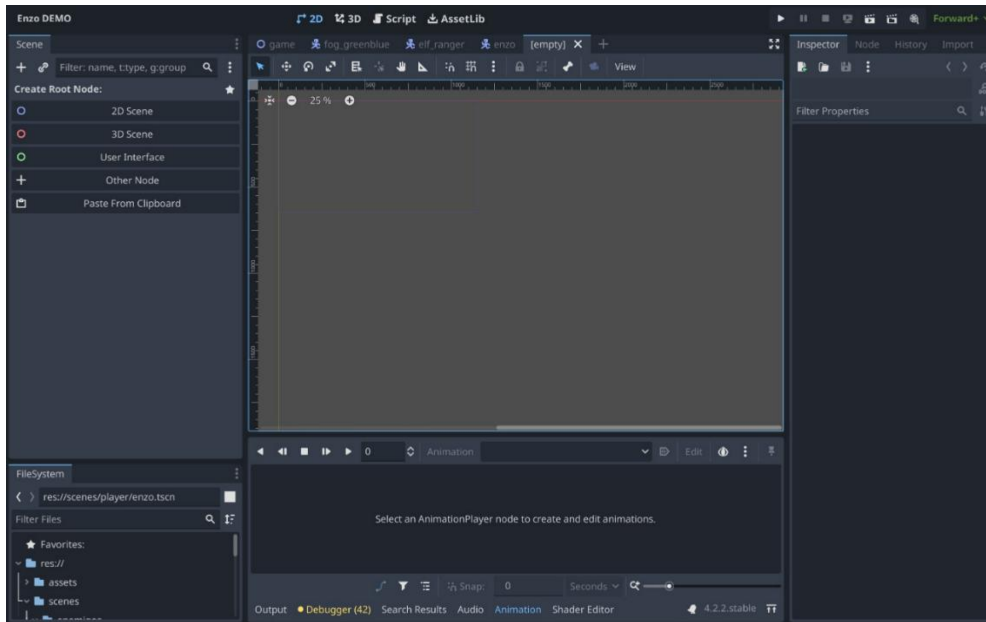
Exportación multiplataforma

Una de las grandes ventajas de Godot es su capacidad de exportar juegos a diversas plataformas, incluyendo **Windows, macOS, Linux, Android, iOS y HTML5**, sin necesidad

de licencias adicionales. Esto lo hace una opción viable tanto para desarrolladores independientes como para estudios profesionales (docs.godotengine.org, 2025).

Pequeña vista a la ventana de GODOT 4

Figura 1. Ventana principal de Godot



Es importante que se tenga un acercamiento inicial al motor, como pueden observar es intuitivo para desarrolladores, tenemos apartados con características diferentes que se hacen notar, además están posicionados de forma que se complementan para agilizar el trabajo, lo principal que se conozcan conceptos como:

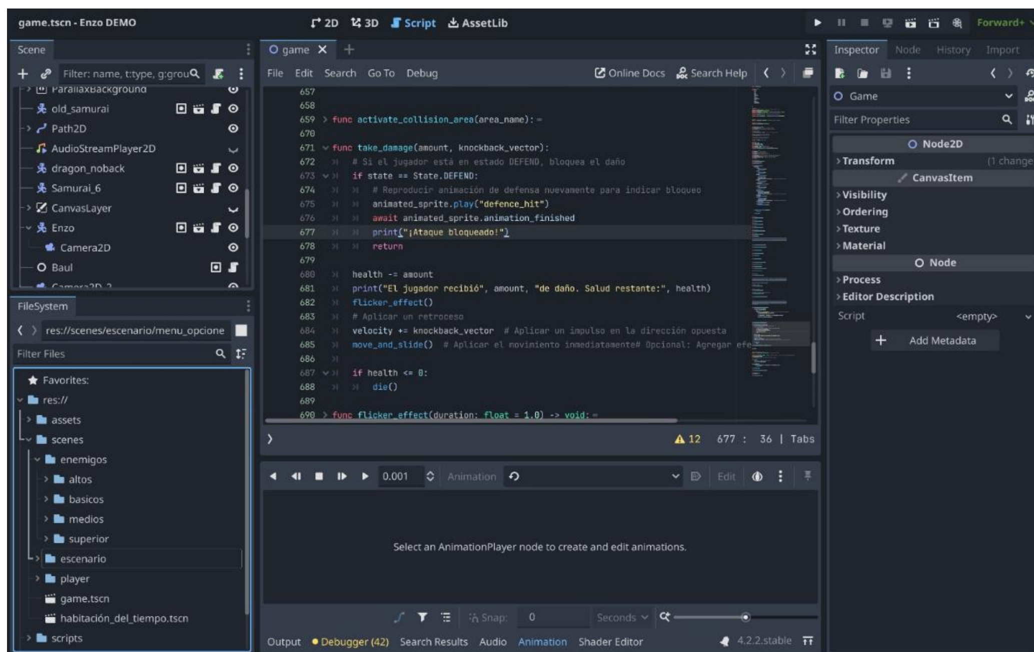
Scene: que muestra todos los nodos que componen la escena.

Inspector: Muestra las características del nodo seleccionado.

BottomPanel: Quizá el más difícil de localizar porque no tiene un nombre como tal en la pantalla, pero es el panel que está en la parte inferior entre el Inspector y Scene.

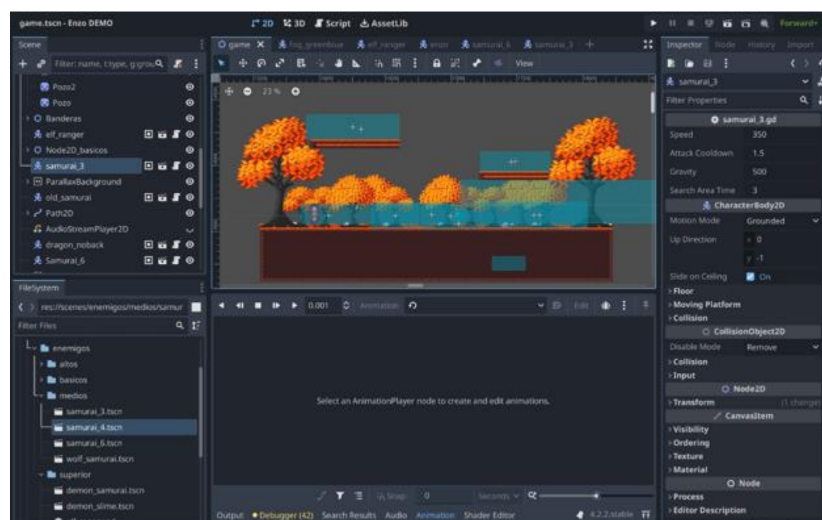
Script: En la parte superior media de la captura podemos ver 4 elementos, estos elementos cambian la vista del editor, el elemento de script permite tener la vista del código, y se ve así:

Figura 2. Vista de Script en el editor de Godot



2D: Es la vista del editor que muestra el escenario en 2D que se está creando. Es una de las vistas más utilizadas durante el desarrollo. Para mayor claridad, se adjunta una imagen como referencia.

Figura 3. Vista 2D



Justificación de uso de GODOT 4 para este proyecto

Se eligió Godot como motor de videojuegos para este demo principalmente porque es de código abierto y completamente gratuito. Esto significa que cualquier ganancia generada por

los juegos será íntegramente del desarrollador, sin preocupaciones sobre cambios en las políticas de monetización.

También se consideró utilizar Unity, pero hace un par de años surgió una controversia debido a su intención de cobrar una tarifa por cada descarga de los juegos desarrollados con su motor (Redacción EC, 2023). La reacción de la comunidad fue inmediata y negativa, lo que llevó a la empresa a revertir su decisión (Tones, 2023). Sin embargo, este incidente demostró que, al depender de Unity, se estaría sujeto a sus políticas y posibles cambios futuros.

En contraste, con Godot existe la seguridad de que su modelo de negocio no cambiará, lo que permite trabajar con total libertad y sin incertidumbre sobre el futuro de los proyectos.

Aplicación de metodologías ágiles en el desarrollo de software

Las **metodologías ágiles** han demostrado ser herramientas efectivas para desarrollar software usando buenas prácticas (Canosa Ferreiro, A. J., 2024). Para un proyecto individual en el que se busca optimizar tiempos y asegurar avances constantes, es fundamental elegir un marco de trabajo que permita adaptabilidad y control sobre cada fase del desarrollo.

Justificación del uso de Scrum

Dado que este proyecto será desarrollado por una sola persona, se decidió usar **Scrum** como metodología ágil, adaptando sus principios a un entorno de trabajo individual. Aunque Scrum suele aplicarse en equipos, sus valores de organización, planificación y entrega incremental son perfectamente aplicables a un desarrollador en solitario.

Scrum se basa en la **inspección y adaptación constante**, lo que permitirá evaluar el progreso del juego en cada ciclo de desarrollo, realizar mejoras continuas y responder rápidamente a posibles cambios en los requisitos. Al utilizar Sprints definidos, se podrán establecer objetivos claros en cada iteración y asegurar que cada etapa del desarrollo tenga un resultado funcional.

Razones clave para elegir Scrum:

- **Gestión estructurada del desarrollo**: Permite dividir el trabajo en iteraciones manejables y medibles.
- **Entrega incremental**: Cada Sprint generará un avance concreto en el videojuego, evitando retrasos innecesarios.
- **Adaptabilidad a cambios**: Si surgen mejoras o ajustes en las mecánicas o diseño, pueden implementarse sin afectar el progreso global.
- **Organización personal**: El uso de tableros Scrum y planificación estructurada facilitará la administración del tiempo y las tareas.

Adaptación de Scrum para un solo desarrollador (Scrum Guide, 2025)

Si bien Scrum está diseñado para equipos, sus principios pueden ajustarse a un flujo de trabajo individual. En este caso, las principales adaptaciones serán:

Roles adaptados:

1. **Product Owner y Desarrollador**: Al solo haber un único responsable, se asumirán ambas funciones, definiendo las características del juego y ejecutando su desarrollo.
2. **Scrum Master (autogestión)**: La disciplina y autoevaluación jugarán un papel clave para mantener un ritmo de trabajo constante y eficiente.

Eventos en un entorno individual:

- **Planificación del Sprint**: Antes de cada ciclo de desarrollo, se definirán las funcionalidades o mejoras que se implementarán.
- **Autoevaluaciones diarias**: En lugar de reuniones de equipo, se realizarán revisiones diarias para monitorear el progreso y detectar obstáculos.
- **Revisión del Sprint**: Cada Sprint culminará con una evaluación del avance logrado y ajustes para el siguiente ciclo.

Artefactos utilizados:

- **Product Backlog**: Lista organizada de todas las funciones y mecánicas planificadas para el videojuego.
- **Sprint Backlog**: Selección de tareas a completar en cada Sprint.
- **Incremento del Producto**: Cada avance en el videojuego representará un hito tangible dentro del desarrollo.

Estructuras de datos y algoritmos en videojuegos

Los algoritmos y estructuras de datos son esenciales para mejorar el rendimiento y la eficiencia en los videojuegos. Permiten optimizar la navegación de personajes, la toma de decisiones y la detección de colisiones.

Estructuras de datos y su aplicación en videojuegos

Las estructuras de datos son modelos que permiten organizar y gestionar información de manera eficiente. En informática, estas estructuras facilitan la recuperación y manipulación de datos, lo que es crucial en el desarrollo de software y videojuegos. Su correcta implementación permite optimizar el almacenamiento, clasificación y acceso a la información, asegurando un mejor rendimiento del sistema (Universitat Carlemany, 2024).

Importancia de las estructuras de datos en videojuegos

En el desarrollo de videojuegos, las estructuras de datos son clave para mejorar la eficiencia en la gestión de recursos, como la carga de escenarios, la administración de inventarios o la simulación de comportamientos de los NPCs. Además, su uso permite reducir el consumo de memoria y optimizar los tiempos de respuesta del juego.

Tipos de estructuras de datos y su utilidad en videojuegos

- **Arrays**: Almacenan datos de manera ordenada y homogénea, facilitando el acceso rápido a elementos. Se utilizan en videojuegos para manejar listas de elementos como inventarios o animaciones.
- **Pilas (LIFO):** Permiten gestionar eventos y acciones secuenciales en el juego, como el historial de movimientos de un personaje.
- **Colas (FIFO):** Se usan en sistemas de inteligencia artificial y procesamiento de tareas, asegurando que los eventos se ejecuten en el orden en que fueron recibidos.
- **Listas enlazadas**: Facilitan la manipulación dinámica de elementos sin necesidad de definir previamente su tamaño, lo que es útil para gestionar enemigos en un escenario o estructuras cambiantes en el juego.
- **Grafos**: Representan conexiones entre diferentes elementos, esenciales en la generación de mapas, pathfinding y lógica de niveles.

- **Árboles**: Son utilizados en sistemas de navegación, organización de jerarquías y en la toma de decisiones de la IA de los NPCs .

Operaciones clave con estructuras de datos

Las estructuras de datos permiten realizar diversas operaciones esenciales en el desarrollo de videojuegos:

- **Inserción y eliminación**: Añadir o eliminar elementos en listas de objetos del juego.
- **Búsqueda y clasificación**: Organizar elementos en un inventario o determinar la mejor ruta en un mapa.
- **Optimización de memoria**: Reducir la fragmentación y mejorar el uso eficiente de los recursos del sistema.

Algoritmos clásicos en videojuegos

Algunos algoritmos tradicionales se usan para resolver problemas clave en los videojuegos, como la búsqueda de caminos y la toma de decisiones de los NPCs (Astengo-Noguez & Martínez Elizalde, 2024).

- **Búsqueda en profundidad y en anchura:** Métodos que exploran todas las posibles rutas en un escenario. La búsqueda en profundidad sigue un camino hasta el final antes de probar otro, mientras que la búsqueda en anchura revisa todas las opciones de manera sistemática.
- **Búsqueda con poda en árboles de solución:** Reduce las opciones innecesarias dentro de un conjunto de posibilidades, eliminando caminos irrelevantes para mejorar la eficiencia.
- **Máquinas de estados finitos:** Modelos que estructuran el comportamiento de los NPCs, permitiéndoles cambiar de estado según reglas predefinidas.

Comportamientos de movimiento

Los steering behaviors permiten que los personajes se desplacen de forma natural y dinámica, sin depender de rutas fijas. Estos algoritmos calculan el movimiento según la posición, velocidad y objetivos del personaje dentro del entorno del juego. Se utilizan

para mejorar la fluidez de los desplazamientos y evitar movimientos robóticos o poco realistas (Rico Zambrana, 2016).

Tipos principales:

- **Evasión**: El personaje ajusta su trayectoria para alejarse de amenazas, como enemigos o proyectiles, mediante cálculos en tiempo real.
- **Persecución**: Permite a un personaje seguir un objetivo en movimiento, prediciendo su trayectoria para optimizar el seguimiento y evitar colisiones.
- **Deambular**: Genera movimientos aleatorios que evitan patrones repetitivos y predecibles, dotando al personaje de un desplazamiento más natural. Se basa en la modificación de la dirección y velocidad en intervalos controlados para simular un comportamiento autónomo.
- **Alineación y Cohesión**: Son utilizados en simulaciones de grupos de personajes, como bandadas de pájaros o multitudes, donde los agentes ajustan su posición para moverse en conjunto sin chocar entre sí.
- **Llegada y Flee**: "Llegada" ajusta la velocidad de un personaje a medida que se acerca a su destino, mientras que "Flee" permite que un personaje escape de un área con rapidez si detecta peligro.

Estos comportamientos pueden combinarse para crear acciones más complejas y realistas, haciendo que los NPCs reaccionen dinámicamente a su entorno sin necesidad de predefinir rutas estáticas (Rico Zambrana, 2016).

Pool de objetos

El pool de objetos es un patrón de diseño que mejora el rendimiento en videojuegos al evitar la creación y destrucción constante de objetos. En lugar de generar nuevas instancias cada vez que se necesitan, los objetos se crean por adelantado y se almacenan en un grupo, desde donde pueden reutilizarse cuando sea necesario (Unity, 2024).

Este enfoque reduce la carga en la CPU y la memoria, optimizando el uso de recursos y evitando los efectos negativos de la recolección de basura. Cuando un objeto ya no es necesario, en lugar de destruirlo, se desactiva y se devuelve al grupo para su reutilización. Esto es particularmente útil en juegos donde se generan múltiples instancias repetitivas, como proyectiles, partículas o enemigos (Unity, 2024).

La reutilización de objetos mejora la estabilidad del juego al reducir la fragmentación de memoria y los picos de procesamiento causados por la creación y eliminación de elementos. En motores como Unity, donde la administración de memoria es clave para el rendimiento, la implementación de un sistema de pooling contribuye significativamente a mantener una ejecución fluida y sin interrupciones (Unity, 2024).

Principios de interacción humano-computador

El diseño de videojuegos no solo requiere mecánicas sólidas y gráficos atractivos, sino también una interfaz de usuario (UI) clara y una experiencia de usuario (UX) fluida. La UI abarca elementos visuales e interactivos que facilitan la navegación del jugador, mientras que la UX influye en la percepción y satisfacción de la experiencia de juego (Talent Garden, 2023).

UI y UX como elementos narrativos

La UI y la UX no solo guían al jugador, sino que también contribuyen a la construcción de la historia. Un diseño adecuado de la interfaz permite que la narrativa fluya de manera natural sin necesidad de explicaciones extensas, favoreciendo la inmersión y facilitando la toma de decisiones dentro del juego (Cerdán, 2023).

Consistencia en la interfaz de usuario

Una **interfaz coherente** ayuda a los jugadores a comprender cómo funciona el sistema, reduciendo la curva de aprendizaje. Se recomienda mantener un diseño homogéneo en botones, etiquetas y colores para mejorar la navegación y evitar confusión (Mahdavi, 2011).

Uso de colores en la UI

El **color** es un recurso visual que debe usarse con precaución, ya que su percepción varía entre usuarios y dispositivos. Para mejorar la accesibilidad, se recomienda emplear una paleta limitada y utilizar gráficos o etiquetas adicionales para diferenciar elementos, asegurando así una interfaz más clara y legible (Mahdavi, 2011).

Reducción de latencia en la respuesta del sistema

Minimizar la **latencia** en las respuestas del sistema mejora la fluidez del juego. Para acciones rápidas, se recomienda una retroalimentación visual inmediata, como animaciones

breves. Para tiempos de espera más largos, los mensajes de estado o indicadores de progreso ayudan a mantener la experiencia sin interrupciones (Mahdavi, 2011).

Uso de metáforas en la UI

Las **metáforas visuales** facilitan la comprensión de la interfaz al asociar funciones con elementos familiares. Sin embargo, es importante que sean comprensibles para todos los jugadores y se usen de manera uniforme en todo el juego. En algunos casos, una función directa puede ser más efectiva que una metáfora visual compleja (Mahdavi, 2011).

Inteligencia artificial aplicada a videojuegos

Comportamiento de enemigos

Los enemigos en los juegos pueden pensar y reaccionar según lo que hace el jugador. Usan sistemas llamados **FSM** o **Árboles de Decisión** para moverse y atacar de manera lógica.

Movimiento de personajes

Los personajes que no controla el jugador, llamados **NPCs**, necesitan encontrar caminos en el juego sin chocar con obstáculos. Para esto, usan métodos como **A*** y **Dijkstra**, que les ayudan a moverse de manera eficiente.

Aprendizaje y mejora

La **inteligencia artificial** puede aprender y cambiar su comportamiento según lo que haga el jugador, haciendo que el juego se vuelva más desafiante y divertido.

Aprendizaje por refuerzo en videojuegos

El **aprendizaje por refuerzo** es un método en el que la IA aprende probando diferentes opciones y viendo cuáles funcionan mejor. Si toma buenas decisiones, recibe una "recompensa", y si se equivoca, recibe una penalización. Con el tiempo, aprende qué hacer para obtener mejores resultados (Amazon Web Services, 2024).

Este tipo de aprendizaje es útil en videojuegos porque permite que los personajes y enemigos se adapten y mejoren su rendimiento, incluso si al principio cometen errores.

Conceptos clave del aprendizaje por refuerzo

Para entender cómo aprende la IA en un videojuego, hay algunos conceptos básicos:

- **Agente:** Es el personaje o enemigo controlado por la IA.
- **Entorno:** Es el mundo del juego donde se mueve el agente.
- **Acción:** Son los movimientos o decisiones que toma la IA.
- **Estado:** Es la situación actual del juego.
- **Recompensa:** Es lo que la IA gana o pierde dependiendo de sus decisiones.
- **Recompensa acumulada:** Es la suma de todas las recompensas que recibe el agente a lo largo del tiempo (Amazon Web Services, 2024).

Cómo funciona en los videojuegos

El aprendizaje por refuerzo usa un sistema llamado **Proceso de Decisión de Markov**, donde la IA prueba diferentes acciones y aprende de los resultados. Así, puede mejorar su comportamiento con el tiempo, explorando nuevas estrategias y usando las que ya conoce para lograr el mejor resultado en el juego (Amazon Web Services, 2024).

DESARROLLO DEL TRABAJO

Planificación del proyecto utilizando Scrum

La planificación del proyecto con Scrum es un proceso iterativo que organiza el desarrollo en ciclos cortos denominados sprints, permitiendo una adaptación continua y una mejora constante. Este marco de trabajo es ideal para proyectos como el desarrollo de un prototipo de videojuego, ya que facilita la gestión de tareas y la integración progresiva de funcionalidades. A continuación, se detalla el enfoque de planificación seguido en el proyecto.

Roles en Scrum

Para simular un entorno profesional y aplicar los principios de Scrum, se asignaron los siguientes roles:

- **Product Owner (PO):** se definió la visión del producto y las funcionalidades clave del prototipo.
- **Scrum Master:** encargado de supervisar el cumplimiento de los principios ágiles y asegurar el progreso constante.
- **Development Team:** responsable del desarrollo técnico y la implementación de las tareas.

Creación del Product Backlog

El Product Backlog es una lista priorizada de tareas necesarias para completar el proyecto. Cada elemento del backlog se describe como una "user story" que incluye descripción, criterios de aceptación y una estimación de esfuerzo. Algunos ejemplos de elementos del backlog para este proyecto incluyen:

- **Movimiento del personaje principal:** Diseñar e implementar el sistema de movimiento.
- **Física del juego:** Configurar las interacciones de colisión y gravedad.
- **Diseño del nivel inicial:** Crear el mapa y las plataformas básicas.
- **Enemigos con IA básica:** Implementar comportamientos simples de ataque y patrullaje.
- **Interfaz de usuario (UI):** Diseñar una interfaz intuitiva y visualmente atractiva.

Planificación de Sprints

El desarrollo del proyecto se dividió en sprints de una semana, con entregables claros al final de cada uno. Inicialmente la duración total del desarrollo se estimó en cuatro sprints, distribuidos de la siguiente manera:

- **Sprint 1:** Configurar el entorno de desarrollo y crear el movimiento básico del personaje.
- **Sprint 2:** Diseñar el nivel y programar las mecánicas de plataformas.
- **Sprint 3:** Incorporar enemigos y desarrollar su lógica de inteligencia artificial.
- **Sprint 4:** Integrar la interfaz de usuario, sonido y ajustes finales.

No obstante, el proceso, junto con su documentación, se extendió hasta un total de diez sprints.

Daily Scrum

Durante cada sprint, se llevó a cabo una sesión diaria breve para reflexionar sobre el progreso. Las preguntas clave fueron:

- ¿Qué se logró el día anterior?
- ¿Qué se planea hacer hoy?
- ¿Existen obstáculos que impidan el avance?

Esta práctica permitió mantener un enfoque claro y resolver problemas de manera ágil.

Revisión y Retrospectiva del Sprint

Al final de cada sprint, se realizó una sesión de revisión para evaluar los avances desarrollados hasta el momento. Además, se llevó a cabo una retrospectiva para analizar lo que funcionó bien, identificar áreas de mejora y planificar cambios para el próximo sprint.

Herramientas Utilizadas

Para apoyar la implementación de Scrum, se utilizaron las siguientes herramientas:

- Word: Para documentar los avances.

Con todo lo antes definido, en el ANEXO 2 se presentan los sprint y el trabajo realizado diariamente en cada sprint.

Diseño Conceptual y Técnico del Videojuego

El diseño conceptual y técnico del videojuego establece la visión creativa y los fundamentos tecnológicos necesarios para su desarrollo. En este proyecto, se abordaron los siguientes aspectos:

Diseño Conceptual

Visión General

El presente trabajo propone el desarrollo de un prototipo de videojuego de plataformas 2D con un enfoque experimental en la implementación de inteligencia artificial (IA). Su característica diferencial radica en la incorporación de un enemigo con aprendizaje por refuerzo, utilizando una Q-table, lo que permite que este aprenda y se adapte en función del entorno.

Propósito

El propósito de este prototipo es evaluar la viabilidad de integrar un sistema de IA basado en aprendizaje por refuerzo simple dentro de un videojuego de plataformas 2D. A través de este experimento, se busca analizar el comportamiento del enemigo y su capacidad de adaptación en tiempo real, brindando una experiencia de juego dinámica y desafiante.

Género y Público Objetivo

El videojuego pertenece al género de plataformas 2D de acción y está dirigido a jugadores interesados en mecánicas dinámicas y desafiantes.

Ambientación y Estilo Visual

El juego está ambientado en un Japón de la era Edo dentro de un mundo de fantasía. Los escenarios presentan una fuerte influencia del otoño, con predominancia del color naranja, reforzando la atmósfera visual característica. En cuanto al arte, el título utiliza un estilo pixel art, lo que proporciona una estética retro que refuerza su identidad visual.

Dentro del mundo del juego, el jugador encontrará samuráis, ranas, dragones y un jefe final representado por un elfo.

Mecánicas y Jugabilidad

El objetivo principal del juego es matar enemigos, recolectar monedas para mejorar el puntaje y llegar al final antes de que se agote el tiempo.

- El jugador comienza en el inicio del mapa y debe llegar al final en un tiempo limitado.
- Existen múltiples enemigos con distintos niveles de dificultad; los más desafiantes otorgan una mayor cantidad de monedas al ser derrotados.

Diseño Técnico

Motor y Plataforma

El videojuego ha sido desarrollado en **Godot Engine 4.2.2** y está diseñado exclusivamente para su ejecución en **PC**.

Arquitectura del Código

- Cada elemento dentro del juego cuenta con un único script asociado a su nodo correspondiente.
- Se ha implementado un **pool de objetos de manera manual** con el propósito de reutilizar instancias de enemigos y otros elementos en la escena, optimizando así el rendimiento.
- El lenguaje de programación utilizado para la implementación del juego es **GDScript**.

Inteligencia Artificial

- Se han incorporado diversas **lógicas simples** para los enemigos, permitiendo que estos ejecuten acciones básicas.
- Conforme aumenta la dificultad del enemigo, su comportamiento se vuelve más complejo, agregando nuevas estrategias y patrones de ataque.
- El sistema experimental de IA basado en **aprendizaje por refuerzo** utiliza una **Q-table**, lo que permite que este enemigo aprenda a reaccionar en función de las condiciones del entorno.

Interfaz de Usuario (UI)

El juego cuenta con diversos elementos de interfaz que garantizan una comunicación efectiva con el jugador:

- **Menús**: Pantallas de inicio, opciones y pausa.
- **HUD** (Head-Up Display): Muestra información esencial durante la partida.
    - **Barra de vida** del jugador.
    - **Temporizador** indicando el tiempo restante.
    - **Contador de monedas** recolectadas.

  - **Contador de shurikens** disponibles.
- Todo el diseño de la interfaz sigue la estética **pixel art**, asegurando coherencia visual con el resto del juego.

Optimización y Rendimiento

Durante las pruebas de rendimiento, se identificó una caída significativa en los **frames por segundo (FPS)** al instanciar múltiples objetos en la escena. Para mitigar este problema, se implementó un **pool de objetos**, lo que permitió una gestión más eficiente de la memoria y los recursos del motor de juego.

- Cada enemigo, al morir, genera **coins** en el escenario.
- Si no se reutilizan las instancias de los objetos, la carga sobre el motor aumenta y la velocidad de procesamiento disminuye drásticamente.
- Gracias a la implementación del **pool de objetos**, se logró reducir la ralentización y mejorar la estabilidad del juego.

Monetización

El videojuego no está destinado a la comercialización. Su desarrollo responde exclusivamente a fines académicos dentro del marco de un proyecto universitario.

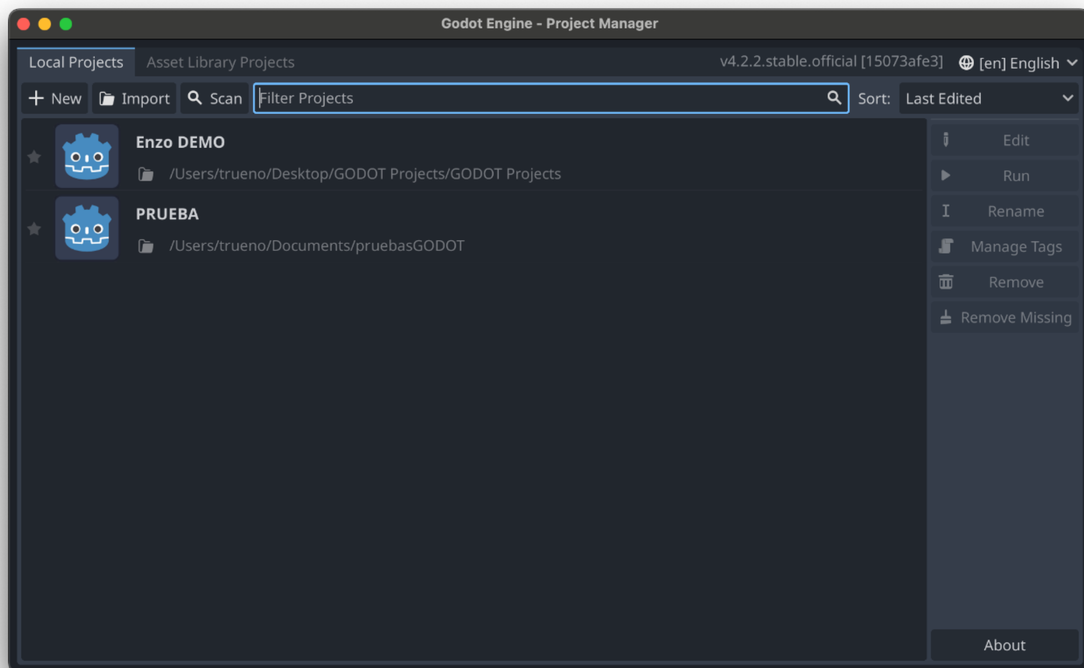
Plan de Desarrollo

El alcance del proyecto se limita a la creación de un **demo funcional**, cuyo propósito es evaluar la implementación de la IA basada en aprendizaje por refuerzo dentro del género de plataformas 2D.

Implementación del prototipo en Godot Engine

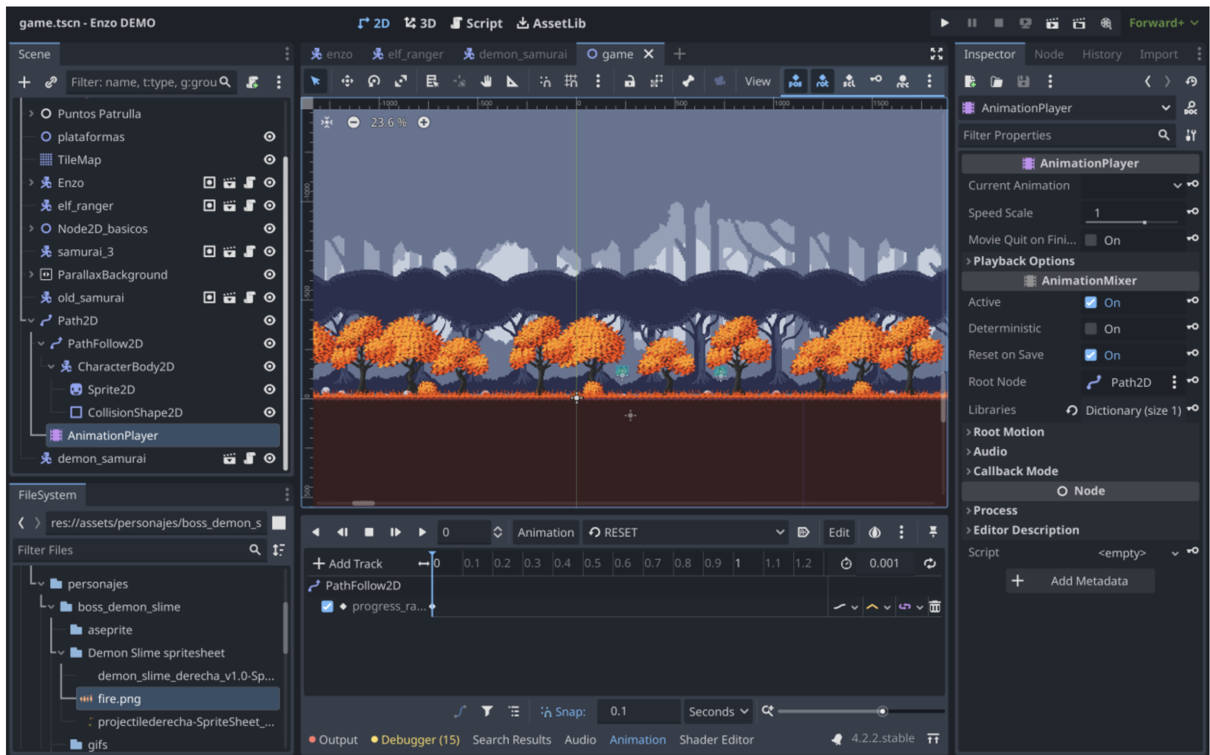
En esta sección se muestra la implementación del prototipo del videojuego utilizando Godot Engine, un motor de desarrollo de videojuegos de código abierto. Se detallarán los componentes esenciales del proyecto.

Figura 4. Ventana tras abrir Godot



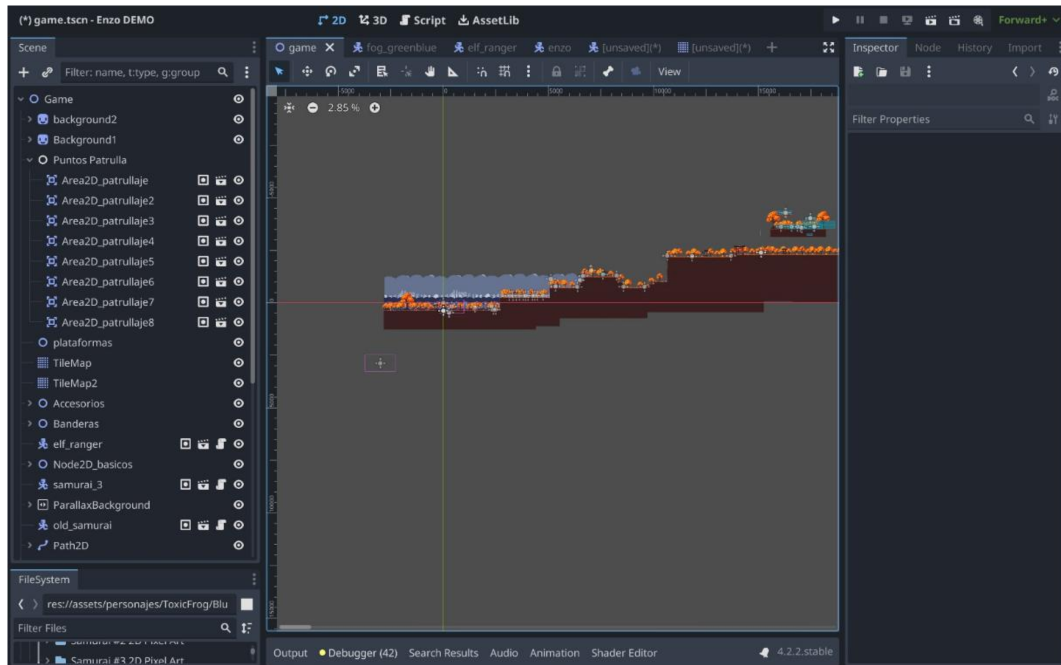
Se puede visualizar la primer ventana tras abrir GODOT, desde aquí se puede editar los proyectos, eliminarlos, ejecutarlos, importarlos, crear nuevos proyectos y similares.

Figura 5. Vista inicial del proyecto en Godot



ELEMENTOS DEL ESCENARIO

Figura 6. Vista alejada de nivel



El escenario está compuesto por varios nodos y elementos clave:

TileMap: Se utiliza para construir el terreno y las plataformas de manera modular.

ParallaxBackground: Implementa un fondo con efecto de desplazamiento para mejorar la sensación de profundidad.

Path2D y PathFollow2D: Se utilizan para manejar el movimiento de ciertos elementos o enemigos a lo largo de un trayecto predefinido.

AnimationPlayer: Permite la gestión de animaciones dentro de la escena.

El escenario no solo sirve como ambientación, sino que también define las áreas jugables y las mecánicas de movimiento del personaje y enemigos. En esta implementación:

- Se utilizan puntos de patrulla que determinan las rutas de los enemigos.
- La configuración de colisiones permite que los personajes interactúen correctamente con el entorno.
- La organización en nodos facilita la escalabilidad y la reutilización de componentes dentro del desarrollo.

Sound: Guarda los nodos que le dan sonido a la escena

Baúl: Es el nodo que permite la reutilización de objetos ya instanciados en la escena.

A continuación algunas imágenes de como se ve el nivel:

Figura 7. Inicio de nivel

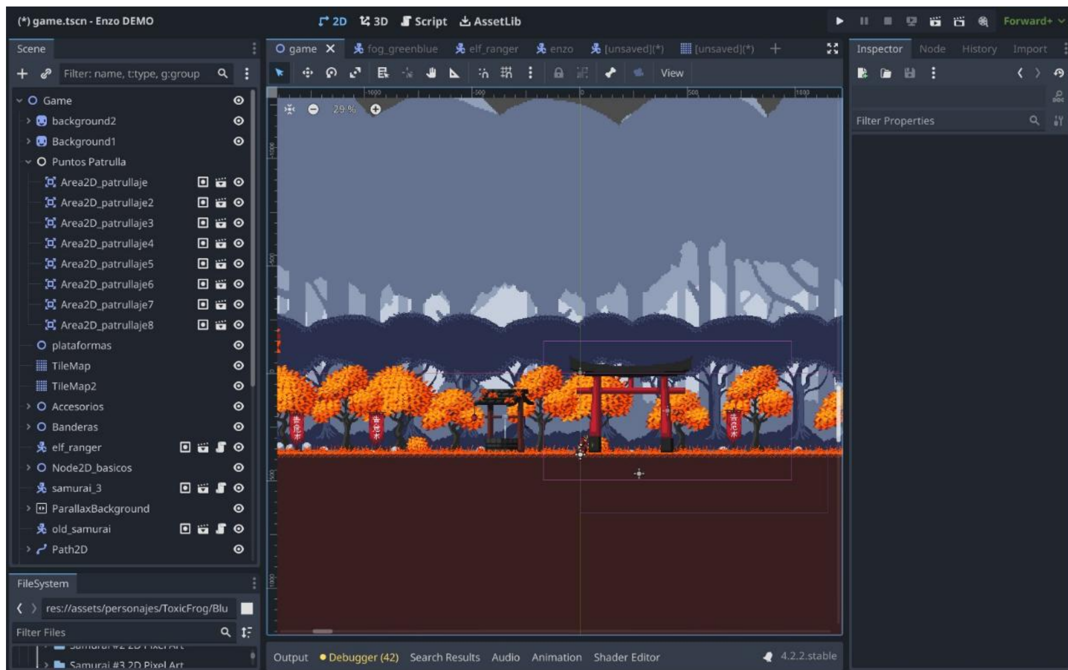


Figura 8. Algunos enemigos en escena

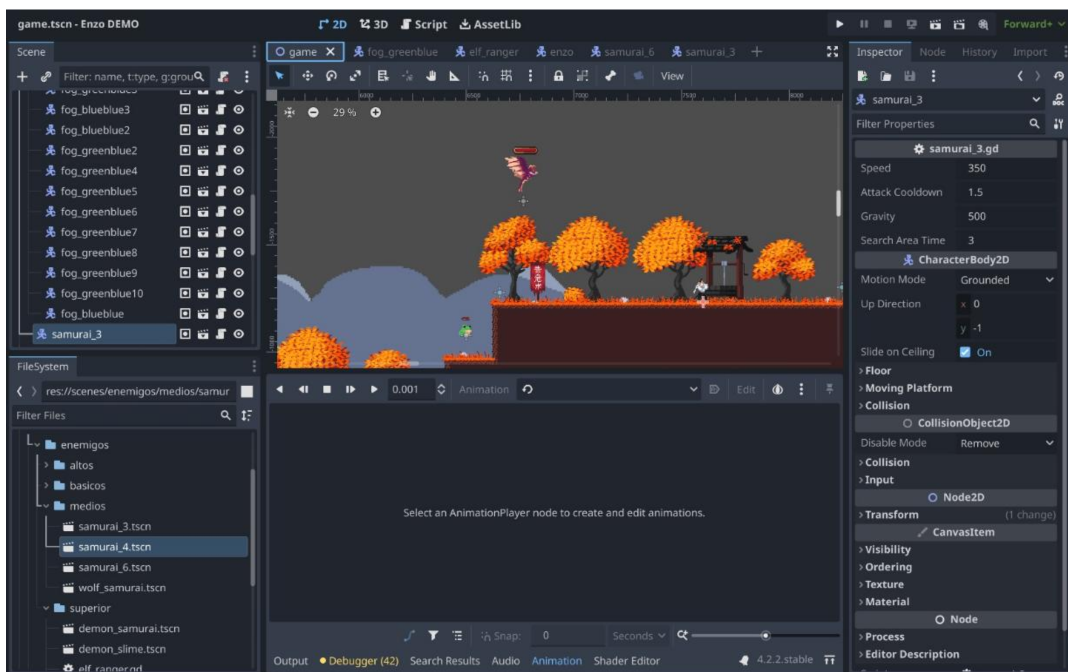


Figura 9. Samurai_6 en escena

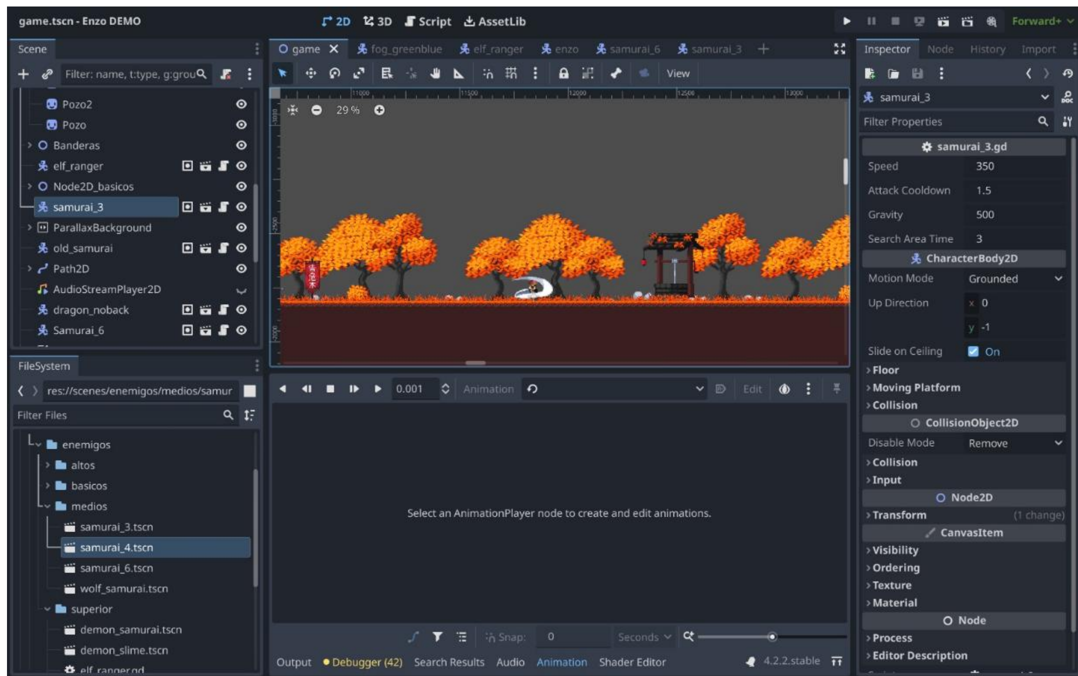
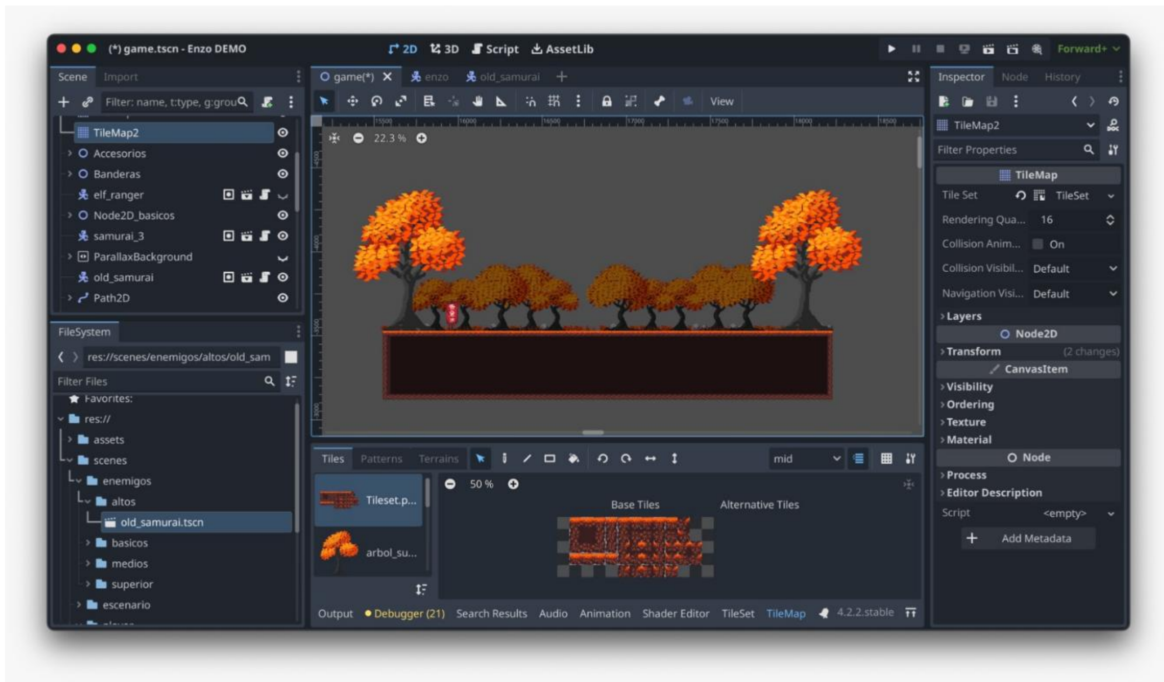


Figura 10. Zona de combate final



MENUS Y TABLAS

Figura 11. Menú inicial

Figura 12. Menú de pausa

Figura 13. Menú de opciones

Figura 14. Menú de controles

Figura 15. Tabla de puntuación

INTERFAZ DE USUARIO (HUD)

Figura 16. HUD Elementos que siempre se muestran en pantalla

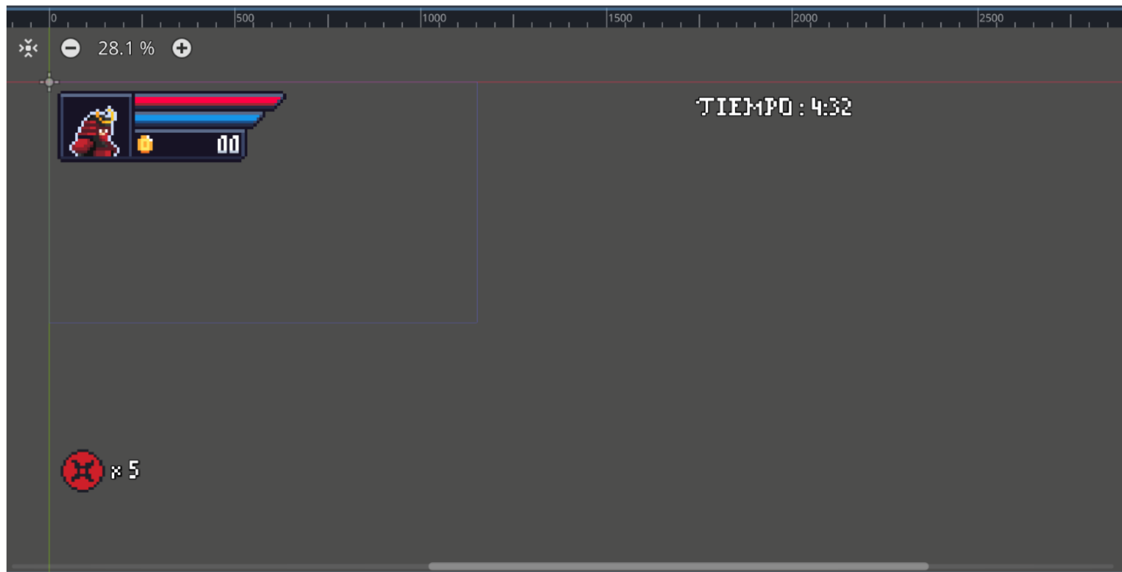
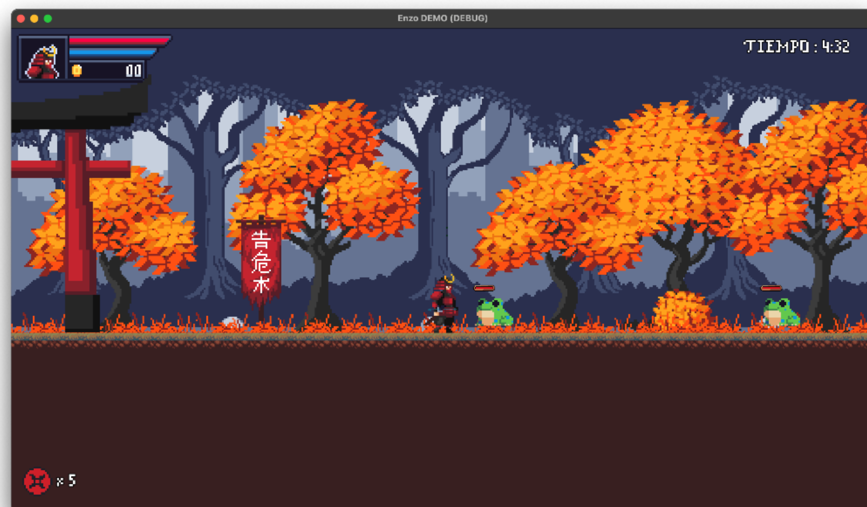


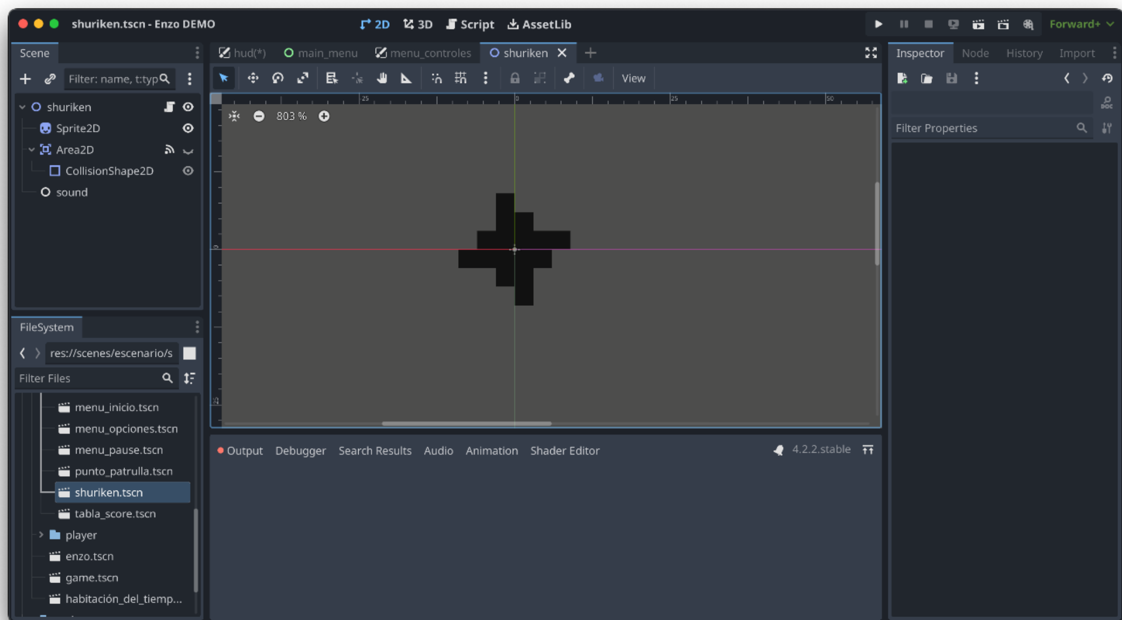
Figura 17. Como se ve en el juego



En la figura 17 se observan los elementos que componen la interfaz de usuario general mientras se juega, muestra el tiempo, los shuriken disponibles, las barras del estado del jugador y los coins que tiene.

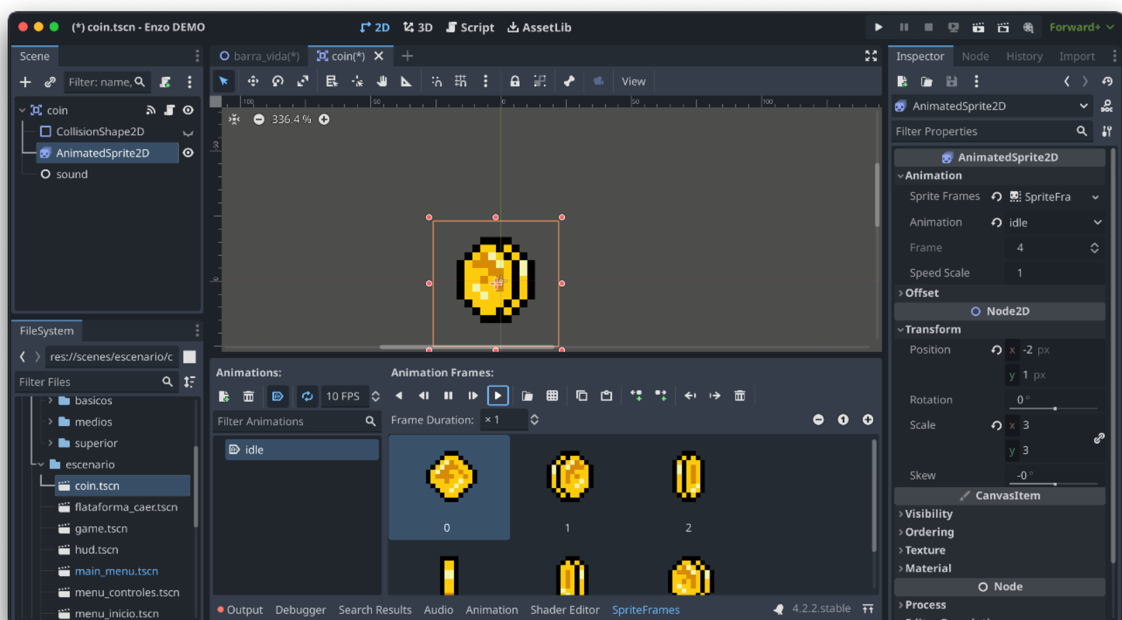
ITEMS

Figura 18. Shuriken



En la figura 18 vemos el item generado cuando el jugador presiona la tecla para arrojar. El item se instancia y sale en línea recta siguiendo la dirección a la que estaba el jugador cuando la lanzó.

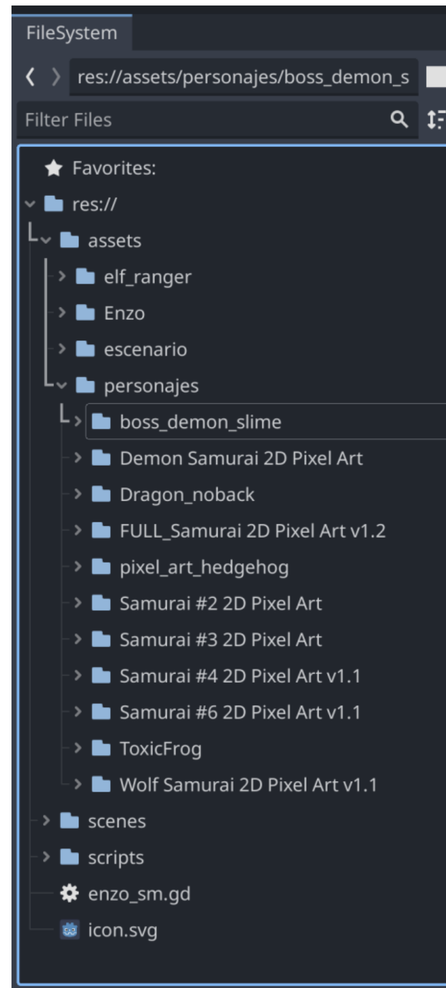
Figura 19. Coins



En la figura 19 se puede observar los coins, elementos que sueltan los enemigos una vez son derrotados, estos están programados para que una vez instanciados, se dirijan hacia el jugador.

ORGANIZACIÓN DE CARPETAS

Figura 20. Directorio

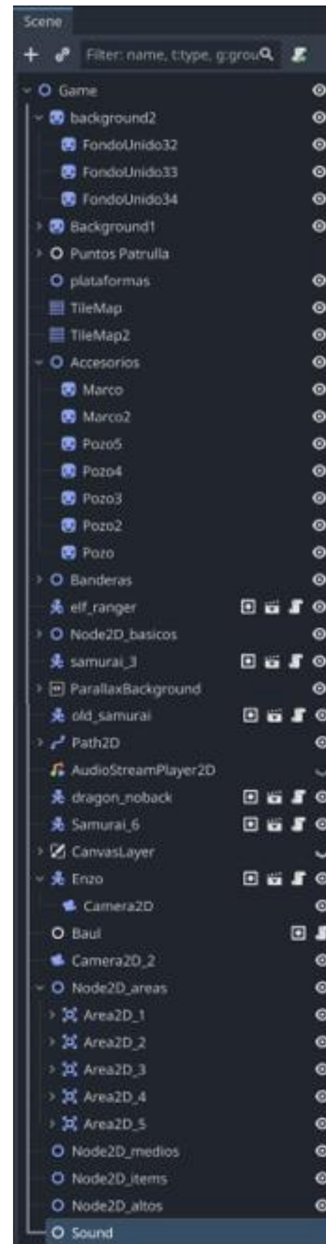


El proyecto se encuentra estructurado en diferentes directorios dentro de la carpeta principal (res://), lo que permite una mejor gestión de los recursos. Algunos de los directorios más importantes incluyen:

- assets/: Contiene los recursos generales del juego.
- scenes/: Almacena las escenas del juego en formato .tscn, organizadas para facilitar la edición y reutilización de niveles y mecánicas.
- scripts/: almacena los códigos.
- scripts/: Carpeta que contiene los archivos de código en GDScript, el lenguaje de programación de Godot.

JERARQUÍA DE NODOS DE LA ESCENA PRINCIPAL

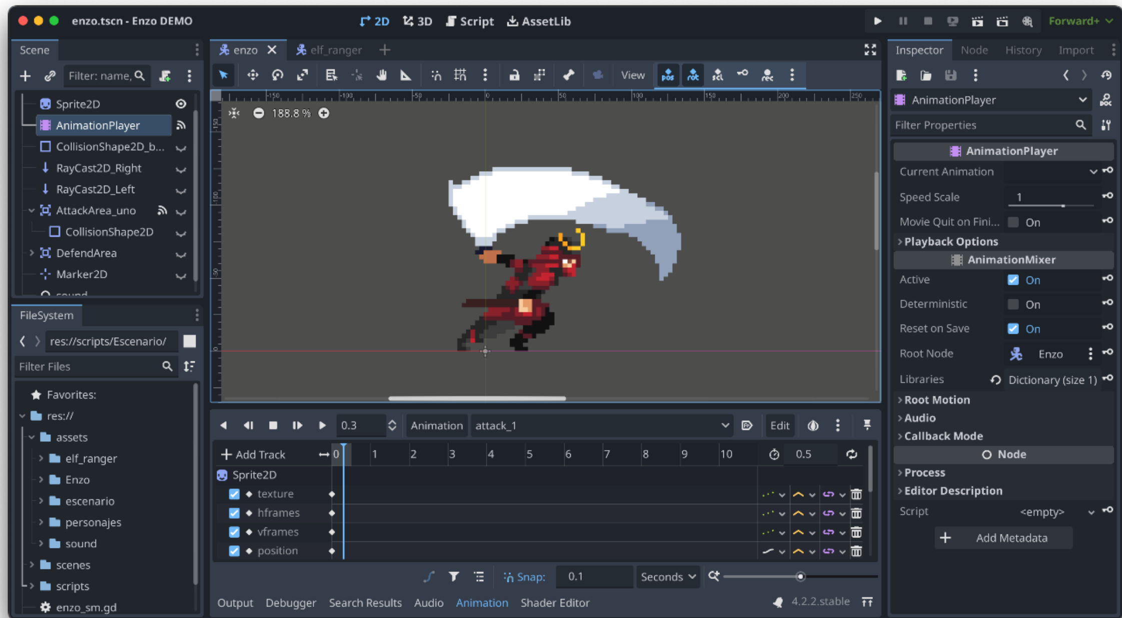
Figura 21. Nodos de la escena principal



En la figura proporcionada se muestra la jerarquía de nodos dentro de la escena principal (game.tscn - Enzo DEMO). En Godot Engine, la estructura de nodos es clave para la organización y funcionalidad del juego. Cada nodo representa un componente del entorno o una entidad interactiva.

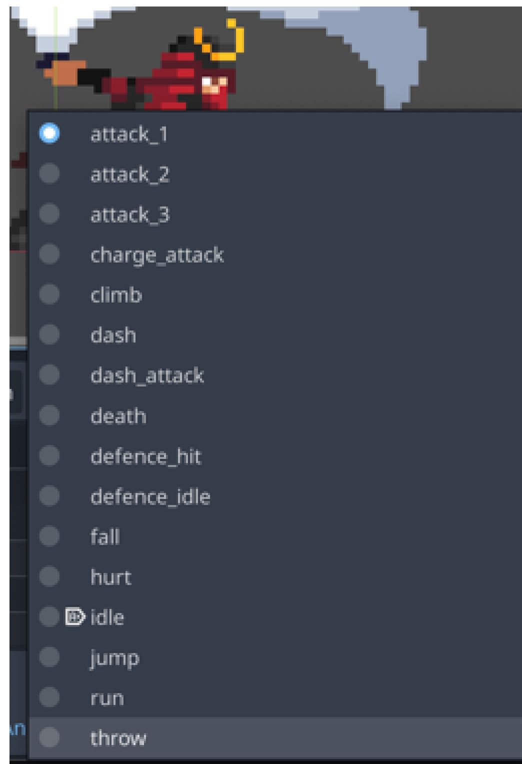
PERSONAJE PRINCIPAL

Figura 22. Personaje Principal - Enzo



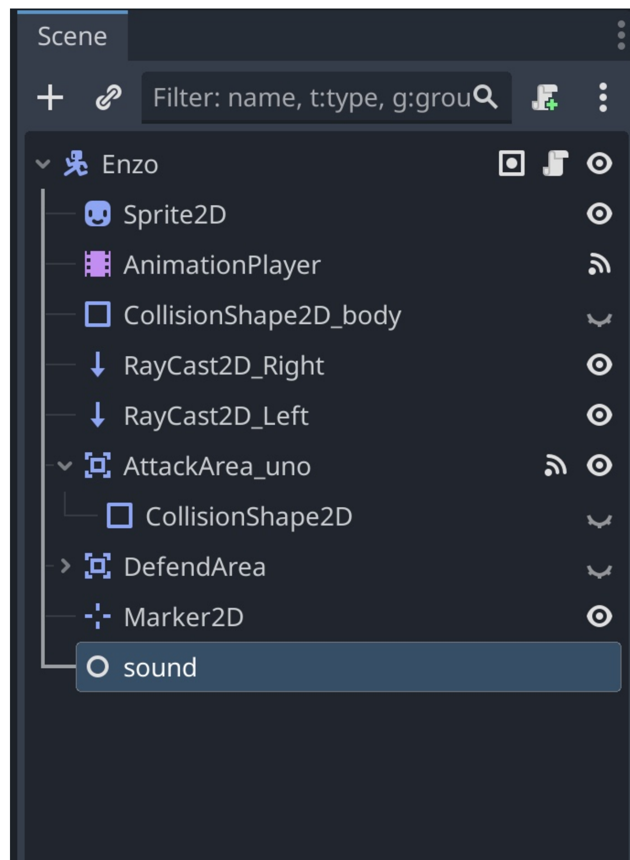
El personaje principal de este prototipo en Godot Engine es Enzo, un samurái con un diseño en pixel art. Enzo es capaz de hacer las siguientes acciones:

Figura 23. Acciones del Personaje



Dentro del nodo Enzo, se encuentran varios subnodos clave que permiten la animación, detección de colisiones e interacciones del personaje:

Figura 24. Nodos de Enzo



Para saber lo que hacen los nodos, se puede visitar el ANEXO 3, donde se presentan todos los nodos de Godot utilizados en este proyecto.

Para ver los algoritmos de los enemigos y objetos se puede visitar el **ANEXO 4**.

IA BÁSICA

En la **IA básica**, los enemigos cuentan con capacidades muy limitadas y poseen un conjunto reducido de movimientos. Generalmente, estos enemigos realizan solo una acción, lo que los hace predecibles y fáciles de enfrentar.

Figura 25. Dragon_noback

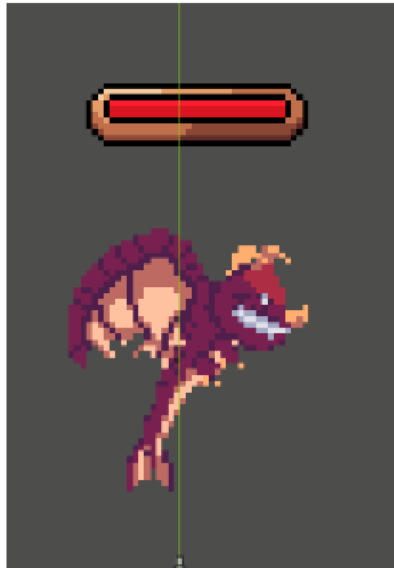


Figura 26. Fog_greenblue

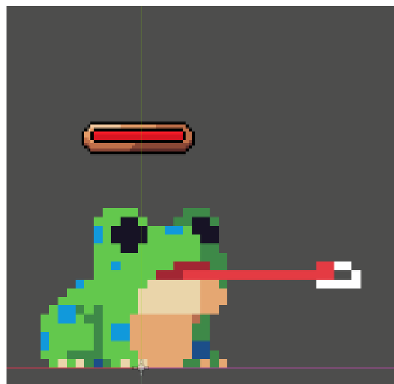


Figura 27. Fog_purpleblue

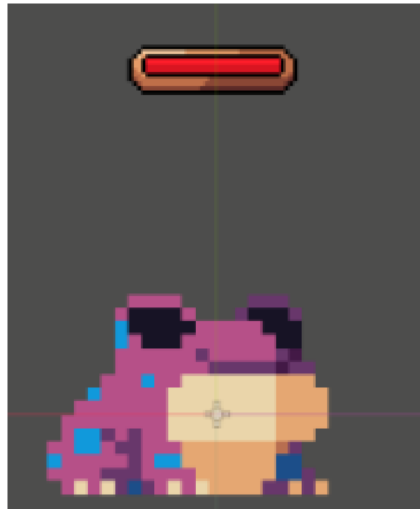


Figura 28. Fog_blueblue

Enemigos tipo rana - Fog Blueblue, / purpleblue / greenblue / Dragon_noback

Estas ranas son enemigos dentro del prototipo en Godot Engine, solo se mueven de un lado a otro, giran cuando tocan algo y cuando tocan al jugador le hacen daño.

Cuando son derrotadas entregan 5 coins.

El dragón tiene un área2d de patrulla, cuando el jugador entra a este área2d el dragón se precipita hacia el jugador y si toca al jugador o el piso explota.

No da monedas porque el jugador no lo derrota.

Estructura de Nodos del Enemigo

Figura 29. Estructura general de ranas

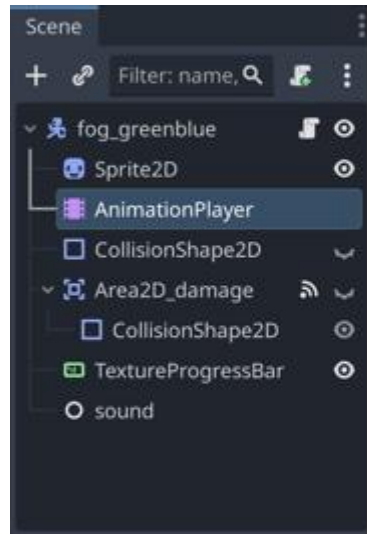
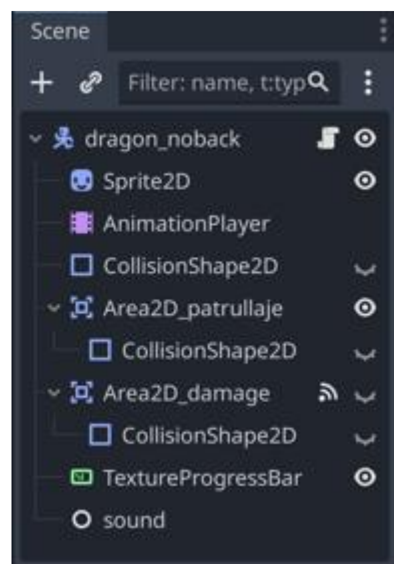


Figura 30. Estructura de dragón_noback



IA MEDIA

En la **IA media**, los enemigos cuentan con capacidades menos limitadas. Tienen movimientos de ataque, patrullan y persiguen al jugador si este ingresa en su zona de patrullaje. Aunque todos comparten el mismo comportamiento, se diferencian por sus animaciones y el área que cubren sus ataques.

Alrededor del mapa hay nodos llamados "puntos patrulla", estas son áreas de colisión que estos enemigos pueden detectar.

Cuando son derrotados entregan 15 coins.

Figura 31. Samurai_6

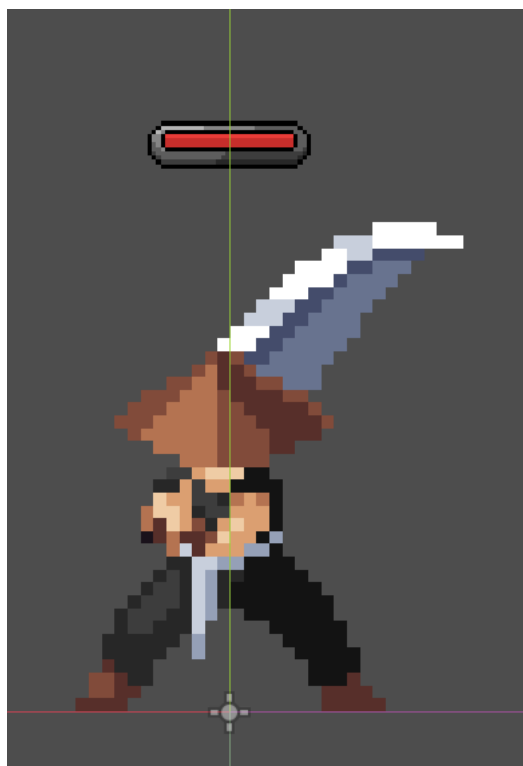
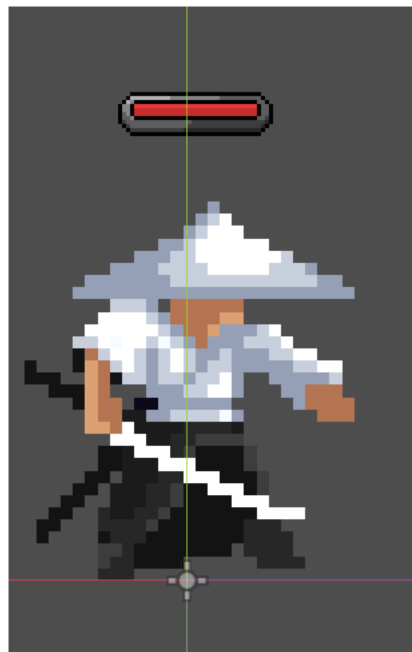


Figura 32. Samurai_4

Figura 33. Samurai_3



Estructura de Nodos del Enemigo

Figura 34. Samurai_6

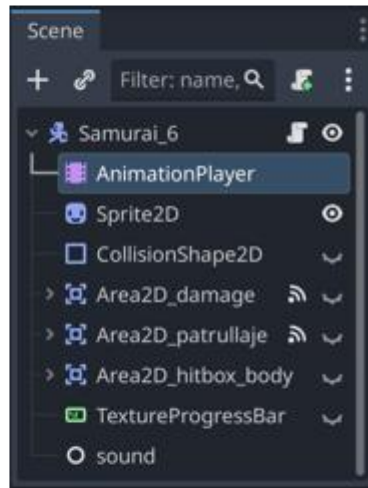


Figura 35. Samurai_4

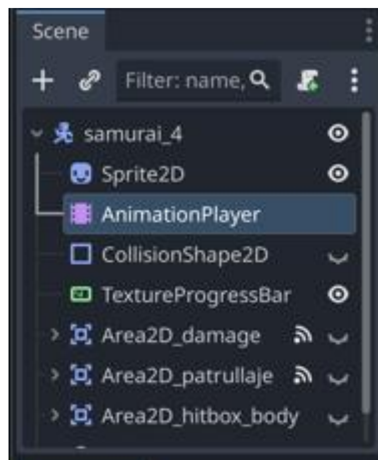
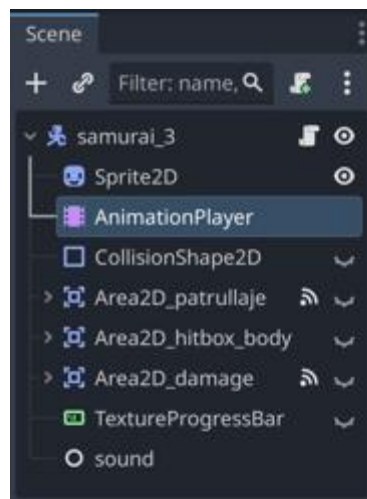


Figura 36. Samurai_3



IA ALTA

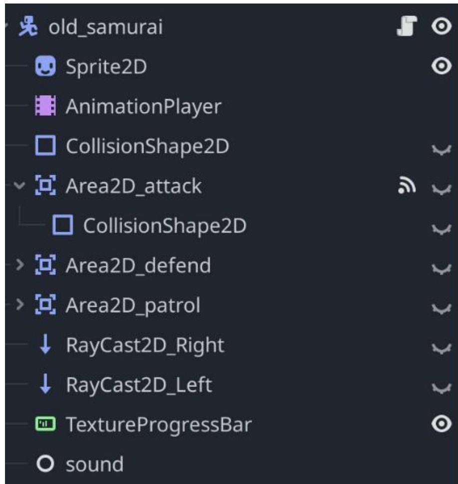
En la **IA alta**, el enemigo cuenta con una variedad más amplia de comportamientos. Además de incluir todas las características de las IA anteriores, como correr, patrullar y atacar al detectar al jugador, este enemigo puede realizar acciones más avanzadas, como saltar, esquivar objetos, defenderse y curarse.

También posee un ataque en *combo* y un ataque especial, lo que lo convierte en un oponente más desafiante. Al ser derrotado, otorga **25 coins** como recompensa.

Figura 37. Old_Samurai

Estructura de nodos del enemigo

Figura 38. Nodos de Old_Samurai

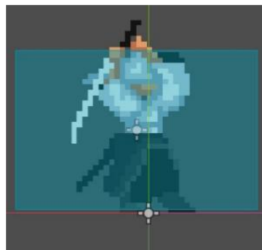


IA SUPERIOR

En la **IA superior**, el enfoque es completamente distinto. A diferencia de las IA anteriores, cuyo comportamiento estaba programado de manera explícita mediante código, esta IA no sigue un conjunto fijo de instrucciones en respuesta a eventos específicos.

Por ejemplo, en la IA media, un samurái ataca cuando el jugador entra en su área de ataque predefinida. En cambio, la IA superior no solo reacciona a eventos programados, sino que aprende y adapta su comportamiento con base en su experiencia dentro del juego.

Figura 39. Area2D de Samurai_3



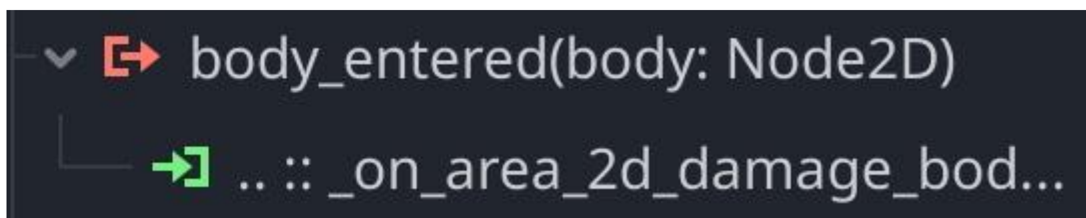
Godot maneja **Signals** (señales), que son un mecanismo de comunicación entre nodos. Estos signals permiten que un nodo envíe notificaciones cuando ocurre un evento específico sin necesidad de una conexión directa entre los nodos.

En este caso, los signals pueden asignarse a un **Area2D**, encargándose de enviar una señal cuando un objeto entra en su área de detección. La señal específica utilizada en este contexto es:

body_entered(body: Node2D)

Esta señal se activa cuando un objeto que pueda colisionar (*body*) entra en el área del nodo Area2D, lo que permite desencadenar acciones como la detección de un jugador, activar trampas o iniciar interacciones en el juego.

Figura 40. Señal body_entered



Esto significa que la señal estará atenta a la entrada de un nodo de tipo **Node2D** (o cualquier nodo que herede de **Node2D)** en el **Area2D** al que ha sido asignada. Si un nodo compatible entra en el área, la función vinculada a la señal se activará automáticamente.

Al crear la señal, se genera la siguiente función (aunque inicialmente vacía):

```
# Detectar colisión con el jugador y aplicar daño
func _on_area_2d_damage_body_entered(body):
    if body.is_in_group("player_group"):
            print("El enemigo tocó al nodo:", body.name)
            var knockback_force = Vector2(200 * direction, -100)
            body.take_damage(damage, knockback_force)
```

En este código, primero se está diferenciando el nodo que ha activado la señal. En este caso, el único nodo que pertenece al grupo **"player_group"** es el jugador, por lo que la función verifica si el nodo que entró en el Area2D es efectivamente el jugador.

Si el nodo detectado pertenece al **"player_group"**, se ejecuta la función take_damage() dentro del jugador, lo que permite aplicarle daño.

Ahora bien, volviendo al tema, las IA's anteriores funcionan de esta forma o de una manera muy similar, estando constantemente a la espera de eventos para activar líneas de código.

Por otro lado, la **IA Superior** trabaja de una forma distinta. Al usar **aprendizaje por refuerzo** (simple en este caso), se le otorga al agente la capacidad de "notar" su entorno y tomar decisiones basadas en su experiencia.

El funcionamiento de esta IA se basa en los siguientes elementos:

- **Agente:** el enemigo controlado por la IA.
- **Entorno:** el nivel en el que se desarrolla la interacción.
- **Conjunto de acciones:** todos los movimientos que el enemigo puede realizar.
- **Estado:** la ubicación o condición en un momento determinado.
- **Recompensa:** un valor positivo o negativo asignado a cada acción realizada.
- **Recompensa acumulada:** el valor total de todas las recompensas obtenidas a lo largo del tiempo.

En este caso, se genera una **q_table** que contiene una matriz de **estado x acción**. Esta tabla se va llenando progresivamente a medida que el agente ejecuta acciones dentro del entorno.

El conjunto de elementos reconocibles en su entorno define su **estado**, y con base en este, el agente toma decisiones. Cada acción realizada, independientemente de si ha dado un buen resultado o no, se registra en la **q_table** junto con su respectiva puntuación.

A través del código, se asignan **recompensas** a las acciones del agente, y este refleja esas recompensas en cada estado de la **q_table**. De esta manera, el agente aprende progresivamente qué acciones le generan mejores resultados, optimizando su comportamiento con el tiempo.

A continuación, se presenta un ejemplo del funcionamiento de la **q_table**. En esta tabla, la primera columna representa el **estado** del agente, mientras que el resto de las columnas corresponden a las **acciones posibles** y sus respectivas **recompensas**. A medida que el agente interactúa con su entorno, cada estado se almacena en la tabla y se actualiza con nuevas experiencias, permitiendo que el agente aprenda de manera progresiva.

El agente toma decisiones basadas en dos estrategias principales: **explotar** y **explorar**. Cuando **explota**, selecciona la acción con el mejor puntaje registrado en la **q_table**, replicando comportamientos que han dado buenos resultados en el pasado. En cambio, cuando **explora**, elige acciones aleatorias con la intención de descubrir opciones que puedan generar un puntaje más alto que las acciones previamente explotadas. Esta combinación de explotación y exploración permite que el agente refine su comportamiento con el tiempo, mejorando su toma de decisiones dentro del juego.

Figura 41. Ejemplo de Q_table

player_in_mid/player_in_back/health/player_state	MOVE_LEFT	MOVE_RIGHT	JUMP	MELEE_ATTACK	DEFEND
1/1/2000/idle	0	0	0	0	0
0/1/1900/run	0	0	0	0	0
1/0/1800/idle	0	0	0	0	0
0/0/1700/moving	0	0	0	0	0
1/1/1600/idle	0	0	0	0	0

El aprendizaje por refuerzo implementado en este proyecto se basa, en esencia, en un enfoque de **fuerza bruta**, donde el agente aprende a través de la prueba y el error. Aunque

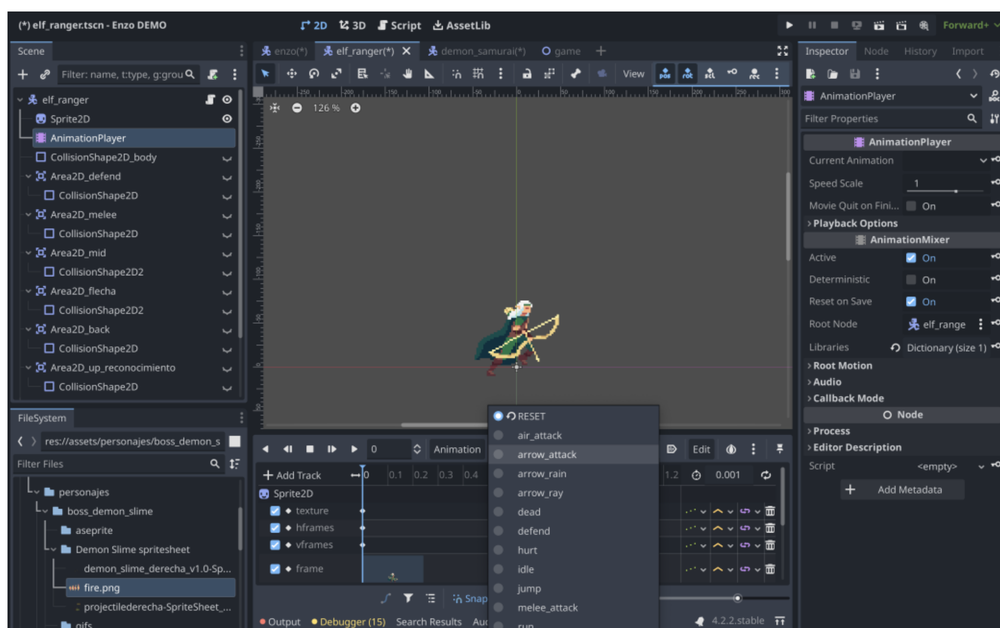
también existe el **aprendizaje por refuerzo profundo**, que utiliza redes neuronales para optimizar la toma de decisiones, su implementación en *Godot* es más compleja debido a la falta de documentación y herramientas especializadas. En comparación con otros motores como *Unity*, la comunidad de *Godot* aún es más reducida, lo que limita el acceso a recursos y ayuda en este ámbito.

Se ha implementado en un solo enemigo, y el algoritmo correspondiente se encuentra en el ANEXO 4.

Los **estados** han ido variando a lo largo del tiempo con el objetivo de optimizar la organización y mejorar el aprendizaje del enemigo. Para ello, se han agregado principalmente **Areas2D** destinadas a la detección del jugador, lo que ha permitido que el enemigo reaccione de manera más eficiente dentro del entorno de combate. Actualmente, el enemigo funciona correctamente en la zona donde fue colocado, atacando con frecuencia cuando detecta la presencia del jugador.

Además de las mejoras en la detección, se han incorporado **Raycast** y **Areas2D** en la escena principal. Estas **Areas2D** se han establecido como estáticas, lo que permite mejorar la interacción del enemigo con el entorno y evitar problemas de detección o comportamiento errático en determinadas situaciones.

Figura 42. Enemigo Principal - Elf Ranger



El Elf Ranger es el enemigo principal del prototipo desarrollado en Godot Engine.

Cuando es derrotado entrega 35 coins.

Estructura de Nodos del Enemigo Principal

Dentro del nodo elf_ranger, se encuentran múltiples subnodos que definen sus mecánicas y comportamiento:

Figura 43. Nodos de elf_ranger

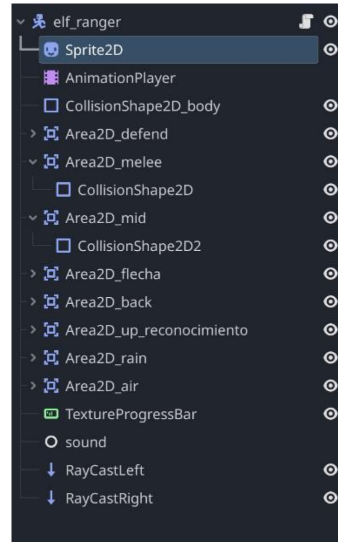


Figura 44. Q_table de elf_ranger

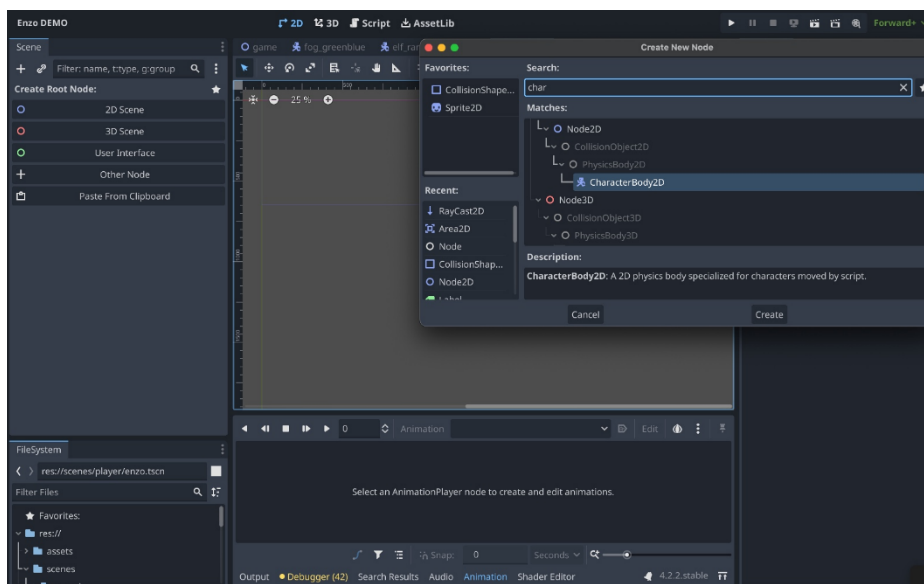
Enzo DEMO — -zsh — 123×41


```
              -7959.85999999999967,
              -12257.72421242554992,
              0,
              0,
              0
        ],
        "[1, 0, 0, 1, 1, 0, 0, 1985, \"JUMP\", 6, false, false, true, false, false, false]": [
              -7478.02000000000044,
              -3935.80000000000018,
              -3976.10000000000082,
              0,
              0,
              -3976.10000000000082,
              0,
              0,
              0
        ],
        "[1, 0, 0, 1, 1, 0, 0, 2000, \"JUMP\", 6, false, false, false, false, true, false]": [
              0,
              0,
              0,
              0,
              0,
              0,
              0,
              0,
              -272.80000000000001
        ],
        "[1, 0, 0, 1, 1, 0, 0, 2000, \"JUMP\", 6, false, false, true, false, false, false]": [
              -230.69999999999999,
              0,
              0,
              0,
              0,
              -230.69999999999999,
              0,
              0,
              0
        ]
}%
trueno@Daniels-MacBook-Air Enzo DEMO %
```

Integración de recursos gráficos y sonoros

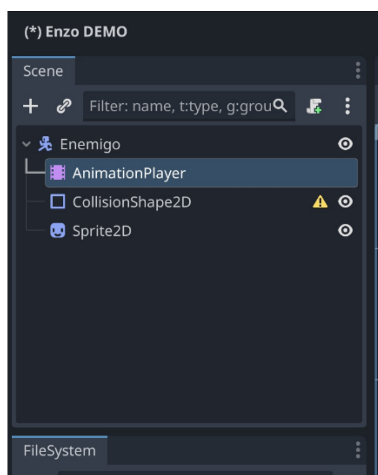
Los recursos gráficos y sonoros son fáciles de implementar en Godot, ya que ofrece muchas facilidades. Como se mencionó anteriormente, todo se maneja a través de nodos, por lo que el sonido y los gráficos también se controlan de esta manera. Asignarlos es un proceso bastante sencillo. Este apartado describe el proceso para asignar recursos gráficos y sonoros en Godot.

Figura 45. Añadiendo un nuevo nodo a la escena (ctrl + a)



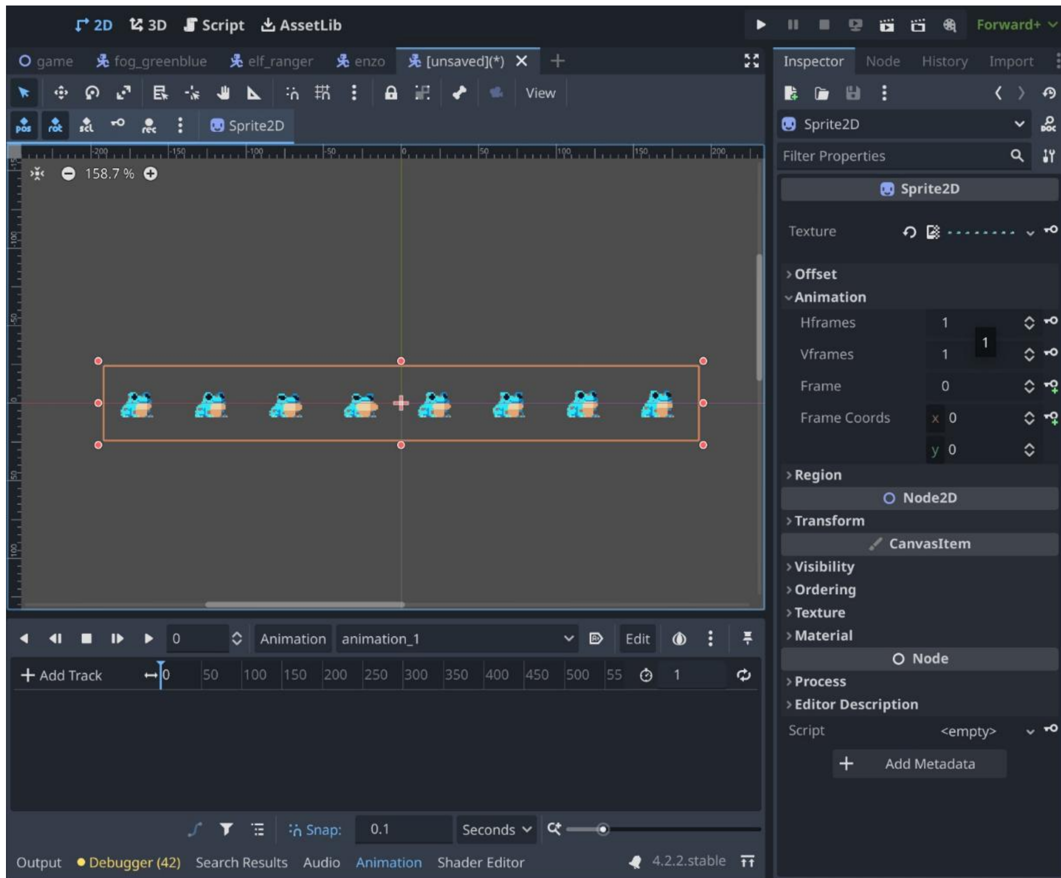
Por ejemplo, en la creación de un nuevo personaje, es necesario generar una nueva escena, lo cual es tan sencillo como abrir una pestaña nueva. Dentro de esta escena, se añaden tres nodos.

Figura 46. Nodos AnimationPlayer, Sprite2D y CollisionShape2D



Una vez añadidos los tres nodos, el inspector del nodo Sprite2D permite asignar la imagen correspondiente en el apartado 'Texture'. Godot ofrece dos opciones para incluir la imagen: arrastrándola directamente al campo o buscándola en el directorio.

Figura 47. Imagen recién cargada en el editor



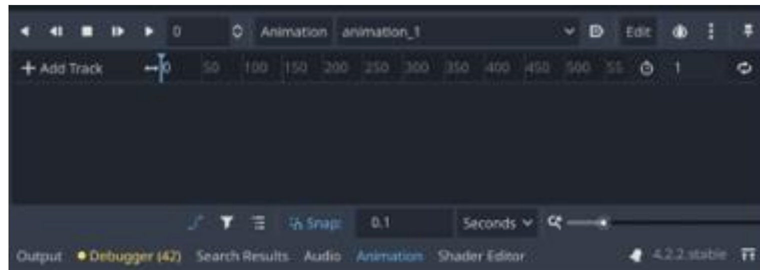
La imagen se mostrará en su totalidad tras la carga. A continuación, es necesario editar la imagen para visualizar únicamente la rana. En el editor, el apartado Animation permite dividir la imagen en secciones.

En los videojuegos 2D, se utilizan imágenes llamadas Sprites, que representan un personaje. Estas imágenes generalmente contienen múltiples acciones y están organizadas en una cuadrícula con espacios entre cada una.

Utilizando las propiedades del apartado Animation, es posible dividir la imagen y obtener los frames individuales. En este caso, la imagen de la rana tiene 8 frames en posición horizontal y 1 en posición vertical. Con esta información, el valor de Hframes (frames horizontales) debe establecerse en 8 y el de Vframes (frames verticales) en 1.

Una vez ajustada la imagen, solo se visualizará un frame de la rana, lo que permitirá animarla. Para animarla, es necesario seleccionar el nodo AnimationPlayer y acceder al apartado Animation en el Bottom Panel. Inicialmente, este panel no contendrá ninguna animación creada.

Figura 48. Animation en BottomPanel



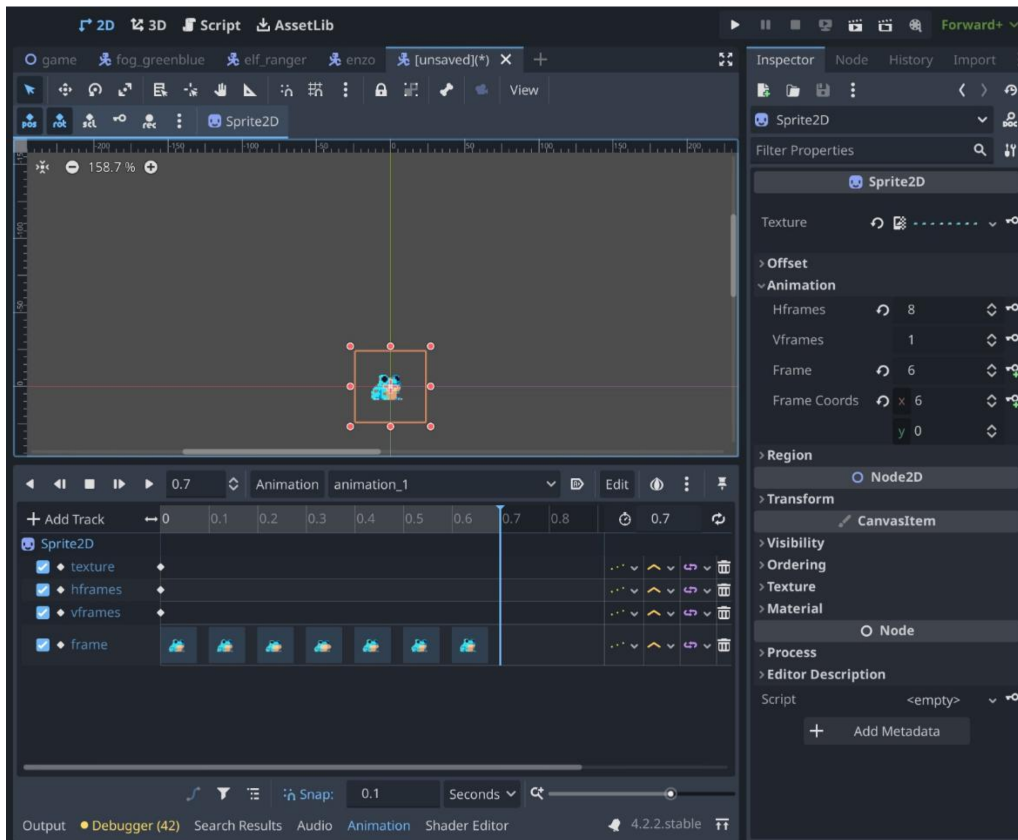
Para animar utilizando el nodo 'AnimationPlayer', es necesario acceder a la parte superior media del 'BottomPanel' y agregar una nueva animación.

Una vez creada la animación, es posible comenzar a configurar el movimiento del personaje. En el inspector del nodo Sprite2D, dentro del apartado Animation, se encuentra un ícono de llaves blancas ubicado al costado izquierdo de las propiedades. Este ícono permite agregar un Keyframe a la animación.

Un Keyframe es un punto de referencia que almacena el estado de un nodo en la línea de tiempo de la animación.

El siguiente paso consiste en establecer en el frame 0 (identificable en la barra superior, donde los números aumentan de izquierda a derecha) los valores de la Textura, Hframes y Vframes. Posteriormente, se debe guardar un Keyframe para cada frame de la rana, resultando en un total de 8 Keyframes, ya que la imagen inicial se dividió en 8 frames que representan cada fase de la animación.

Figura 49. Imagen cortada y animada



En cada secuencia de frames, se agrega un Keyframe para la propiedad correspondiente. Al ejecutar la animación con Play, la rana se moverá según la secuencia establecida.

Este procedimiento permite integrar imágenes en los nodos de Godot de manera eficiente. La misma metodología se aplica a cualquier nodo: al agregar un Sprite2D en el editor, es posible asignarle la imagen deseada.

En cuanto al sonido, su integración sigue un proceso similar al de las imágenes. Sin embargo, antes de incorporarlo, es necesario crear una escena destinada al almacenamiento del sonido.

Figura 50. Nueva escena a partir de un nodo tipo "AudioStreamPlayer2D"

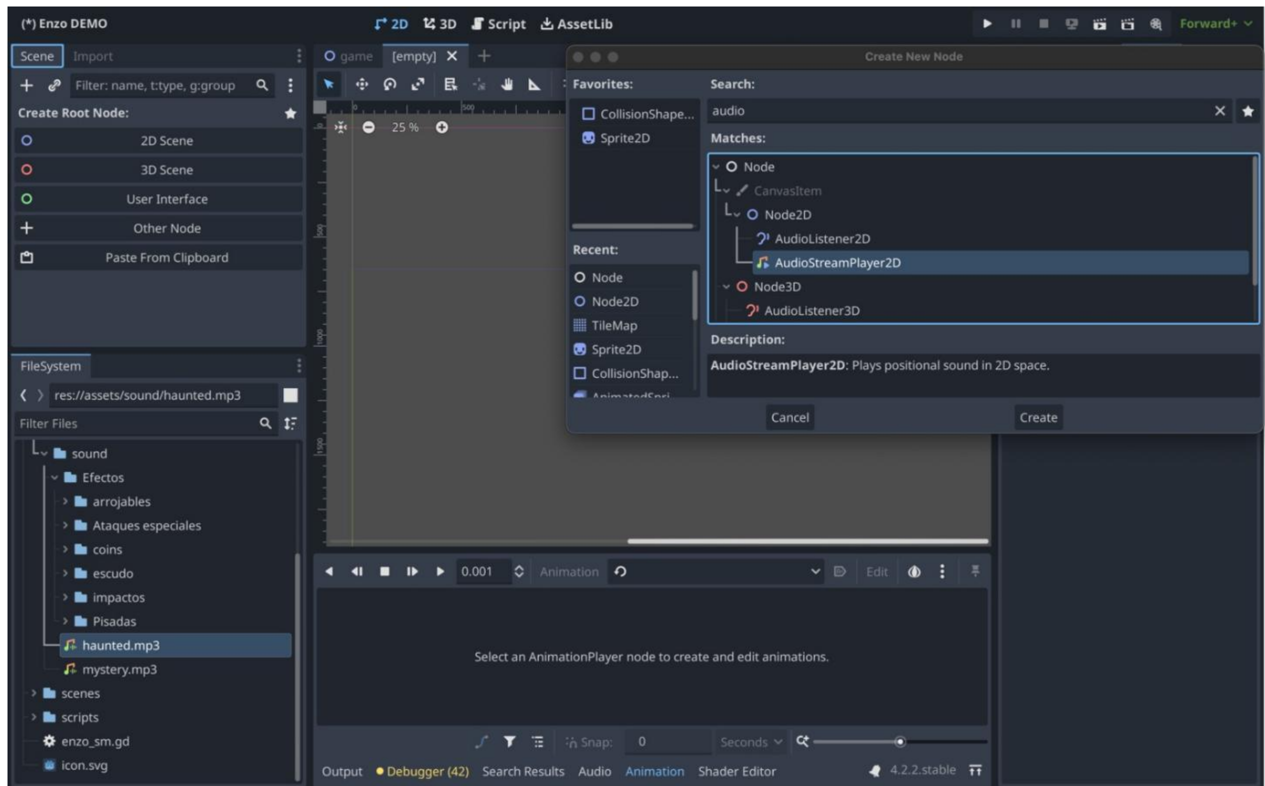
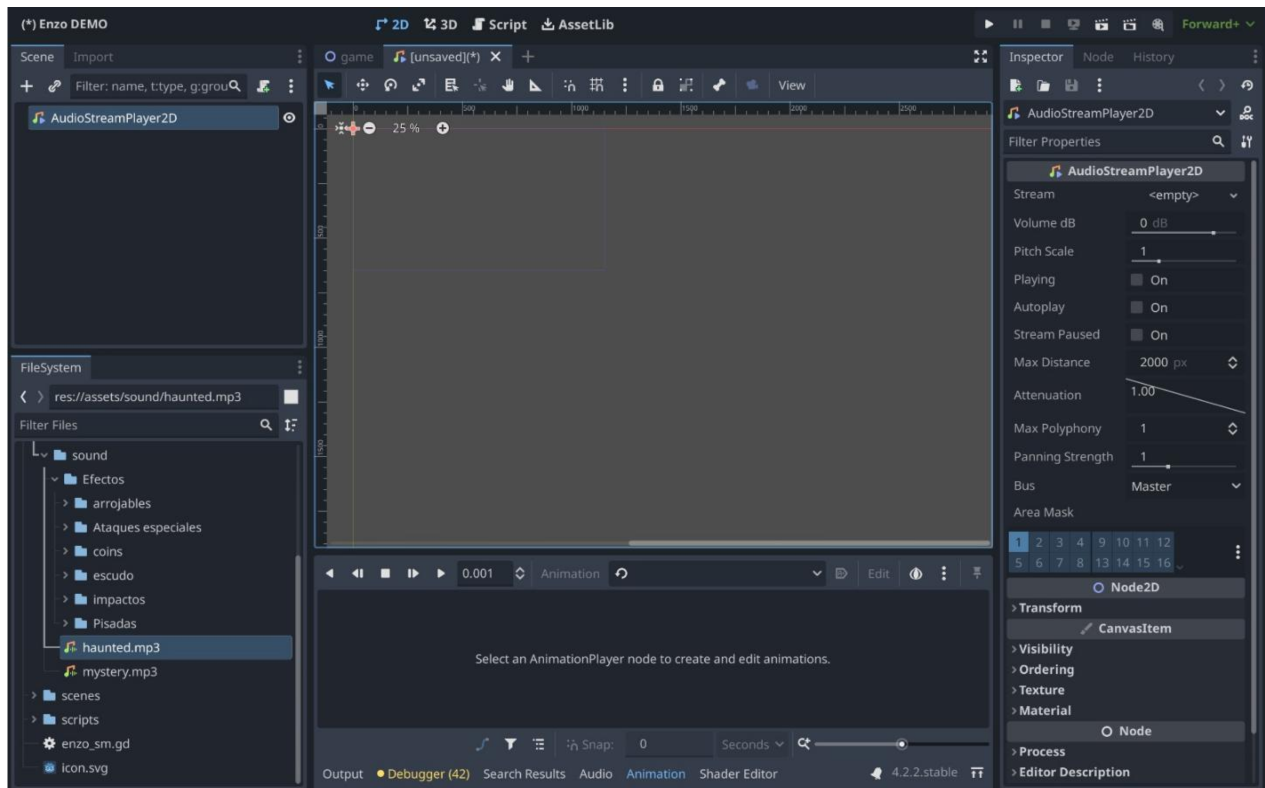
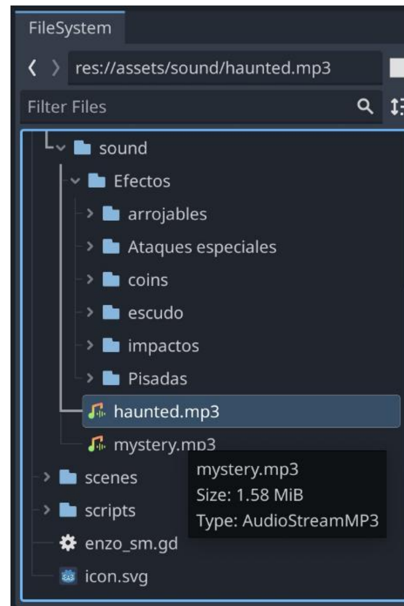


Figura 51. Escena creada con el nodo principal para Audio



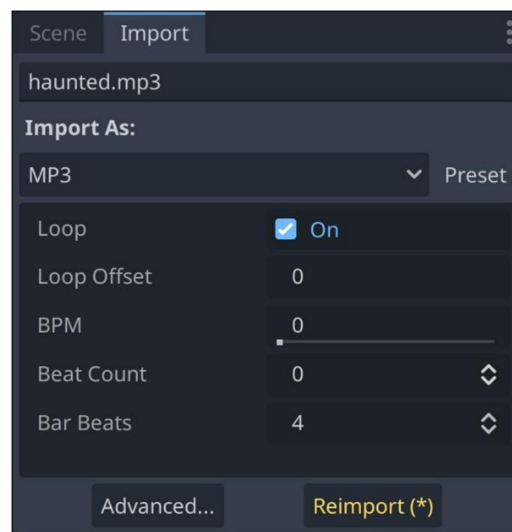
Después de crear la escena, la pista de sonido debe estar preparada para su integración.

Figura 52. Pista de audio haunted



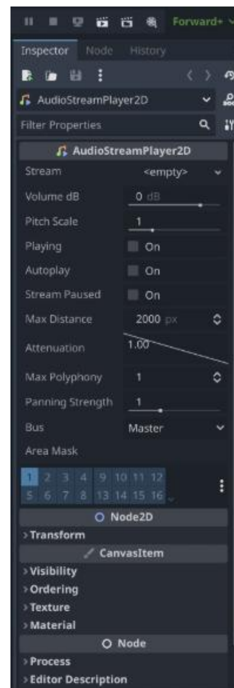
Para importar un recurso en Godot, es necesario acceder a la pestaña Import en el editor.

Figura 53. Import con haunted seleccionado



Para que la pista se reproduzca en bucle, es necesario reimportar el sonido y activar la opción Loop.

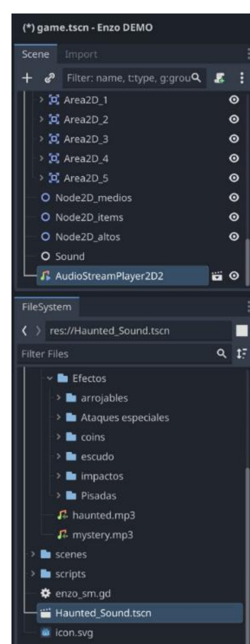
Figura 54. Inspector del nodo "AutioStreamPlayer2d"



Después de completar este proceso, es necesario acceder al inspector del nodo principal y asignar el sonido a la propiedad Stream, como se muestra en la imagen anterior.

Una vez agregado, la escena debe guardarse para disponer de un nodo de sonido reutilizable en otras escenas. Para integrarlo en una nueva escena, el nodo puede arrastrarse directamente al entorno deseado, asegurando su incorporación al proyecto.

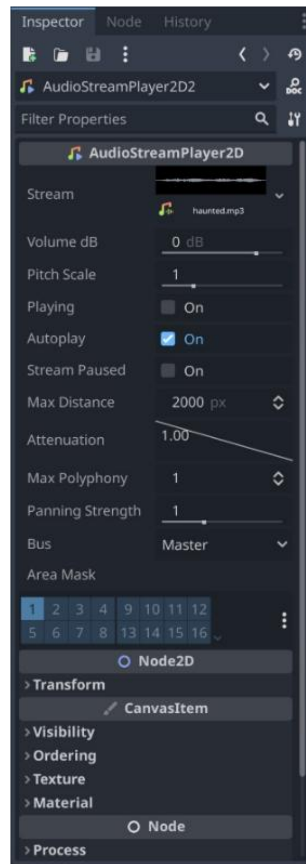
Figura 55. Nodo AudioStreamPlayer2D agregado a la escena



JESUITAS ECUADOR

El nodo Haunted_Sound, correspondiente a la escena recientemente creada, puede añadirse a otra escena. Su gestión se realiza a través de sus propiedades en el inspector.

Figura 56. Propiedades de "AudioStreamPlayer2D"



Estos serían todos los pasos, ahora la música sonará en la escena donde has colocado el nodo.

Análisis de Resultados

Pruebas internas y validación del prototipo

El juego ha sido sometido a un proceso de prueba exhaustivo con el objetivo de identificar y corregir cualquier error que pudiera afectar la jugabilidad, la estabilidad y el rendimiento del sistema. Durante estas pruebas, se evaluaron diversos aspectos, incluyendo la mecánica de juego, la respuesta de los enemigos, el posicionamiento de los elementos en el escenario, la detección de colisiones y la optimización del rendimiento.

Se llevaron a cabo pruebas en distintos escenarios y condiciones, simulando tanto situaciones comunes como casos extremos considerando la escalabilidad del proyecto.

Gracias a este proceso, se identificaron y corrigieron errores que afectaban la experiencia del usuario. A continuación, se presentan los principales problemas detectados junto con las soluciones implementadas:

1. **Caída del jugador fuera del mapa**
    - Problema: En ciertas áreas, el jugador podía caer fuera del mapa al ser atacado por un enemigo debido a la falta de colisiones.
    - Solución: Se agregaron colisiones adicionales en las zonas afectadas para evitar este error.
2. **Posicionamiento inadecuado de los enemigos**
    - Problema: Algunos enemigos estaban ubicados en posiciones que los hacían prácticamente imposibles de esquivar, o bien, se encontraban agrupados en exceso en un solo punto.
    - Solución: Se ajustó la distribución de los enemigos para mejorar la jugabilidad y el equilibrio del combate.
3. **Estancamiento de los enemigos en su ruta de patrulla**
    - Problema: Cuando un enemigo perdía de vista al jugador, quedaba atrapado corriendo en una sola dirección, ya que no encontraba un punto de patrulla para redirigir su movimiento.

- Solución: Se reubicó el último punto de patrulla, colocándolo junto a la pared, evitando que los enemigos salgan de los límites de su ruta y asegurando que puedan redirigirse correctamente.

4. **El jefe final (*Elf Ranger*) salía del escenario al saltar**
   - Problema: *Elf Ranger* tenía una capacidad de salto elevada, lo que le permitía salir accidentalmente del escenario.
   - Solución: Se redujo la capacidad de salto para mantenerlo dentro del área de combate.
5. **Caída de rendimiento al instanciar múltiples elementos**
   - Problema: Al instanciar una gran cantidad de elementos en la escena, el rendimiento del juego disminuía drásticamente.
   - Solución: Se implementó un sistema de *Object Pooling* llamado "Baúl". Este nodo gestiona los objetos instanciados para que, en lugar de ser eliminados tras cumplir su función, se almacenen en una zona específica y puedan reutilizarse posteriormente. Esto optimiza el rendimiento y mejora la escalabilidad del proyecto.
6. **Problemas con los Sprites de Old Samurái**
   - Problema: Los *Sprites* de *Old Samurái* estaban mal dimensionados, lo que provocaba errores en su manejo. Por ejemplo, cuando el personaje se movía en otra dirección, el *sprite* se salía del área de colisión. Esto generaba una sensación extraña, ya que, al acercarse el jugador al enemigo, el *sprite* del enemigo se superponía al jugador debido a la falta de una colisión que impidiera este efecto.
   - Solución: Antes del cambio, se utilizaba una sola animación de correr, la cual se modificaba mediante código utilizando la propiedad *flip_h*, que permite voltear horizontalmente el personaje. Este método funcionaba correctamente para otros *sprites* bien dimensionados, pero no para *Old Samurái*.
   - Para solucionar este problema, se decidió crear dos animaciones de correr: una orientada hacia la izquierda y otra hacia la derecha. De esta manera, cuando el

enemigo corre, simplemente se activa la animación correspondiente en lugar de voltear el *sprite*. Esto permite reposicionar el personaje y cambiar directamente de animación, evitando los problemas de colisión y superposición.

Evaluación interna de usabilidad y experiencia de usuario

En este apartado se presentan los recursos de la interfaz de usuario (UI) y su gestión dentro del juego. A pesar de ciertas limitaciones en tiempo y recursos, la experiencia es completamente funcional y práctica, asegurando una navegación intuitiva y accesible para el jugador.

Funcionalidades de la UI

El jugador cuenta con diversas opciones para interactuar con la UI, todas implementadas con el objetivo de mejorar la accesibilidad y la comodidad en la experiencia de juego. Entre las principales funciones destacan:

- **Menú de Inicio:** Desde aquí, el jugador puede iniciar una nueva partida o salir del juego.
- **Menú de Pausa:** Se puede acceder en cualquier momento de la partida, permitiendo al jugador detener el juego temporalmente. Desde este menú, es posible reanudar la partida con el botón "Continuar".
- **Menú de Controles:** Proporciona la opción de modificar las teclas de juego según las preferencias del jugador, facilitando una experiencia más personalizada.
- **Menú de Opciones:** Permite ajustar los controles del juego y silenciar el sonido si así se desea.
- **HUD (Heads-Up Display):** Presenta información esencial en pantalla de manera clara y acorde con el estilo visual del juego. A través del HUD, el jugador puede monitorear:
  - **Barras de estado:** Indicadores que muestran la salud y energía del personaje.
  - **Monedas:** Recuento de la cantidad de monedas recolectadas.
  - **Shurikens disponibles:** Cantidad de proyectiles restantes para su uso.

Experiencia de Control y Movilidad

El sistema de controles ha sido diseñado con una gran atención a la fluidez y precisión, garantizando que el jugador pueda moverse con naturalidad a lo largo del mapa. La capacidad de respuesta es inmediata, lo que permite una ejecución eficiente de acciones sin retrasos

perceptibles. Esto asegura que la experiencia de juego se sienta dinámica y ágil, especialmente en momentos de acción intensa.

Además, el jugador cuenta con una variedad de movimientos que enriquecen la jugabilidad y brindan una sensación de libertad al explorar el entorno. La posibilidad de desplazarse de diferentes maneras contribuye a una mayor inmersión, permitiendo al jugador adaptarse a diversas situaciones dentro del juego. La combinación de estos movimientos con un HUD bien diseñado facilita la toma de decisiones estratégicas en todo momento.

Análisis del cumplimiento de los objetivos planteados

Los objetivos específicos se cumplen a lo largo de los capítulos de este trabajo. Sin embargo, existen otros objetivos planteados en el Marco Teórico, los cuales fueron fundamentales para el desarrollo del proyecto. Estos objetivos son los siguientes:

Estructuras de Datos y Algoritmos

Este apartado es más evidente en el Baúl, el pool de objetos utilizado para optimizar el uso de nodos.

Estructuras de Datos Utilizadas

El código implementa una gestión eficiente de nodos mediante un **Pool de Objetos**, evitando la creación y eliminación constante de instancias. Para lograr esto, se utilizan diferentes algoritmos que permiten clasificar, almacenar y recuperar nodos. A continuación, se analizan los principales procesos algorítmicos que estructuran el código.

Identificación y Clasificación de Nodos

El primer paso en la gestión del pool es **determinar el tipo de cada nodo** según su archivo de escena. Para esto, se obtiene el nombre base del archivo asociado a la escena de cada nodo y se utiliza como clave para almacenarlo en el diccionario de objetos. Si un nodo no tiene un archivo de escena asignado, se descarta inmediatamente. Este procedimiento garantiza que los nodos del mismo tipo sean almacenados juntos, lo que facilita su recuperación y evita inconsistencias en la gestión de objetos.

Este proceso de clasificación funciona como un sistema de **indexación por clave**, permitiendo que cada tipo de nodo tenga su propia estructura de almacenamiento. Gracias a esto, cuando se necesita un nodo de un tipo específico, se puede acceder a su conjunto de objetos de manera directa, sin necesidad de recorrer toda la colección de nodos.

Almacenamiento de Nodos en el Pool

Cuando un nodo se almacena en el pool, el algoritmo sigue una secuencia de pasos que garantizan un uso eficiente de los recursos. Primero, se obtiene su tipo y se verifica si ya existe una categoría en el diccionario de almacenamiento. Si aún no hay una entrada para ese tipo de nodo, se crea una nueva estructura para almacenarlos y se asigna una posición inicial donde serán guardados.

Antes de añadir un nodo al pool, el sistema verifica si ese mismo nodo ya se encuentra en la lista, evitando duplicados. Si el nodo es válido para almacenamiento, se procede a **desactivarlo**, moviéndolo a una zona específica fuera del área visible del jugador. Luego, el nodo se **añade a una pila de almacenamiento**, lo que permite un acceso rápido en futuras recuperaciones. Para mantener el orden, se actualiza la posición donde se almacenará el próximo nodo del mismo tipo, asegurando que todos los objetos de un mismo tipo se guarden de manera estructurada.

Recuperación de Nodos desde el Pool

Cuando se necesita reutilizar un nodo, el algoritmo verifica si hay objetos disponibles en la pila del tipo requerido. Si existen nodos almacenados, se **extrae el último nodo agregado**, asegurando que el sistema siga una lógica **Last In, First Out (LIFO)**. Este enfoque garantiza que los objetos más recientemente almacenados sean los primeros en ser reutilizados.

Una vez extraído, el nodo se **reactiva** y se asigna la nueva posición donde debe aparecer en la escena. Si no hay nodos disponibles en el pool, se devuelve un valor nulo, indicando que no se puede reutilizar ningún objeto en ese momento.

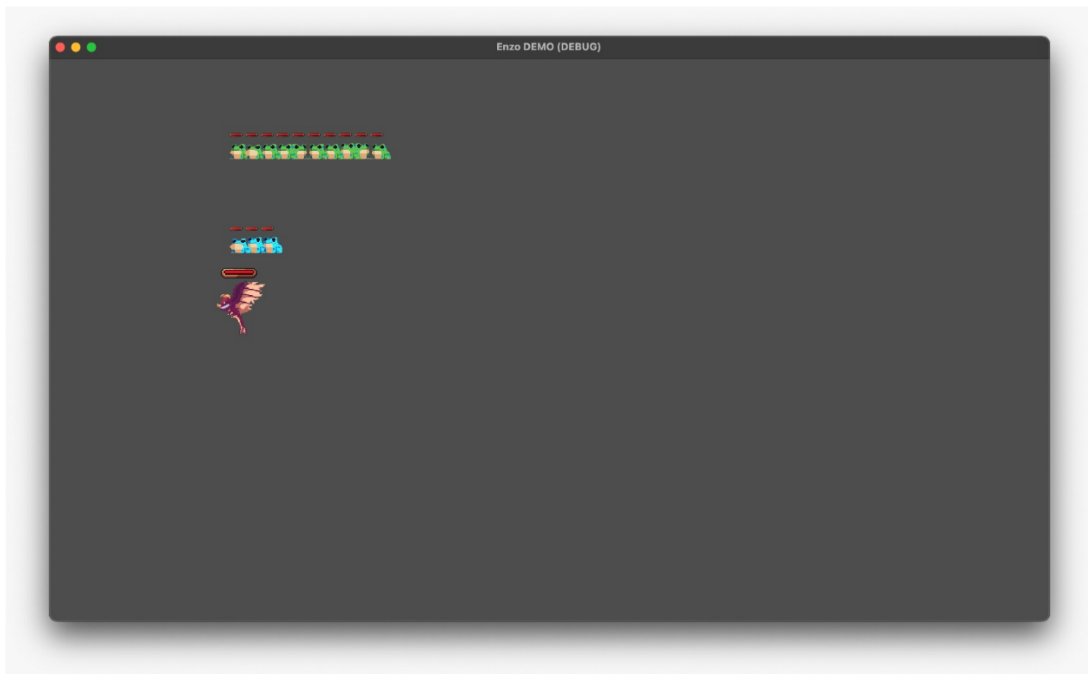
Este mecanismo previene la generación innecesaria de nuevos objetos, lo que optimiza el uso de memoria y mejora el rendimiento del juego.

Estructura Algorítmica y Optimización del Código

El código sigue un diseño algorítmico estructurado basado en diccionarios y pilas para optimizar la gestión de nodos. El uso de **diccionarios** permite acceder rápidamente a los objetos según su tipo, mientras que el empleo de **pilas** para almacenamiento y recuperación garantiza una administración eficiente de los nodos disponibles. La verificación de duplicados previene errores y mantiene la integridad del sistema, asegurando que cada nodo sea almacenado y recuperado correctamente.

Este enfoque demuestra el uso de principios algorítmicos fundamentales en la optimización de videojuegos, aplicando estrategias de clasificación, búsqueda y gestión de memoria para mejorar la eficiencia del código. Al estructurar el almacenamiento y recuperación de nodos como un algoritmo bien definido, se logra un sistema robusto que reduce la carga del procesador y evita problemas de rendimiento relacionados con la creación y eliminación constante de objetos en tiempo de ejecución.

Figura 57. Vista de Baúl en tiempo real.



Ingeniería de Software

El desarrollo del proyecto se gestionó utilizando Scrum, cuya documentación se encuentra en el Anexo 2.

Interacción Humano-Computador

Consistencia visual

Para garantizar una interfaz de usuario coherente, se seleccionaron y adaptaron elementos visuales compatibles con el diseño del juego. Se integraron sprites obtenidos de diversas fuentes y se modificaron en Aseprite para ajustarlos a los requisitos del proyecto.

Los sprites seleccionados fueron ensamblados en la interfaz manteniendo una estética visual homogénea.

Además, se procuró mantener uniformidad en tablas, botones y colores. Dado que no se contaba con un dominio completo sobre el uso del color, se optó por una paleta reducida para evitar inconsistencias, considerando que mantenerlo simple era la mejor opción.

Posicionamiento de la interfaz

El diseño de la interfaz tomó como referencia estándares reconocidos en videojuegos, específicamente la estructura visual de *Hollow Knight*, conocida por su interfaz bien integrada.

El uso de GAME UI DATABASE, una plataforma que recopila interfaces de usuario de diversos videojuegos, permitió facilitar este proceso.

Metáforas visuales

Dado que se trata de un demo con contenido limitado, la cantidad de elementos de la UI es reducida, pero adecuada para ilustrar este apartado.

El uso de metáforas visuales facilita la transmisión de información de manera intuitiva. Algunos ejemplos en el proyecto incluyen:

Shuriken x numero: representa la cantidad de shurikens disponibles. Este ícono tiene dos estados:

- **Color rojo** cuando el jugador tiene al menos un shuriken disponible.
- **Color apagado** cuando no hay shurikens, indicando que la acción no está disponible.

Barras del HUD:

La barra roja indica la salud del personaje, asociándose visualmente con la sangre para representar la vitalidad.

La barra azul representa la resistencia del personaje, regulando el uso del ataque especial y asociándose visualmente con la energía.

Barras de vida de los enemigos:

Las barras de vida de los enemigos varían según su nivel de dificultad: la IA básica usa un color marrón similar a la tierra o el bronce; la IA media emplea un tono gris oscuro evocando hierro o piedra; la IA avanzada presenta un color plateado; y la IA superior incorpora un diseño distintivo con un tono similar al platino.

Este tipo de representaciones facilita la comprensión del jugador sin necesidad de texto adicional, haciendo que la interfaz sea más eficiente y clara.

Retroalimentaciones visuales

Para mejorar la experiencia del jugador, se implementaron efectos visuales que refuerzan la interacción con el entorno. Entre ellos:

- **Animación de daño:** Cuando el personaje recibe un impacto, se activa una animación específica para representar el daño sufrido.

- **Sistema de retroceso:** Cuando un personaje es golpeado, este se desplaza hacia atrás mientras se reproduce la animación de daño.

Estos efectos refuerzan la inmersión y comunican visualmente el impacto de las acciones del jugador dentro del juego.

Inteligencia Artificial

En el desarrollo de videojuegos, la inteligencia artificial (IA) desempeña un papel crucial en la creación de comportamientos realistas y desafiantes para los enemigos controlados por el sistema. En este proyecto, se implementó un enemigo basado en aprendizaje por refuerzo (Reinforcement Learning, RL) utilizando una Q-Table para la toma de decisiones en tiempo real.

Este enfoque permite que el enemigo aprenda estrategias y adapte su comportamiento según la situación del jugador, en lugar de seguir una programación de movimientos predefinida. La metodología empleada se basa en el algoritmo Q-Learning, el cual optimiza la selección de acciones a través de un sistema de recompensas y penalizaciones.

Algoritmo de Aprendizaje por Refuerzo: Q-Learning

El Q-Learning es un algoritmo de aprendizaje por refuerzo que permite a un agente aprender una política óptima para seleccionar acciones en un entorno dado. Se basa en una tabla de valores Q(s, a) que representa la calidad de elegir una acción a en un estado s.

En la implementación de este enemigo, cada estado se representa como una combinación de factores clave del entorno, y las acciones disponibles incluyen movimiento, ataques y defensa.

Modelado del Enemigo y Estados del Entorno

El enemigo está diseñado como un **CharacterBody2D** dentro del motor de juego Godot. Se define un conjunto de **acciones** disponibles para el enemigo, tales como:

- Movimiento a la izquierda/derecha
- Salto
- Ataques cuerpo a cuerpo
- Defensas
- Ataques a distancia (flechas, ataques aéreos, lluvia de flechas)

El estado del entorno está compuesto por información relevante como:

- Posición del jugador respecto al enemigo.
- Si el jugador está dentro de un área de ataque o defensa.

- Si el enemigo tiene visibilidad del jugador.
- Si el enemigo está bloqueado por un obstáculo.
- Salud actual del enemigo y el jugador.

Para representar los estados de manera eficiente, cada estado se codifica en una cadena de texto que agrupa las condiciones del entorno en un formato interpretable por la **Q-Table**.

Selección y Ejecución de Acciones

La elección de acciones sigue una estrategia donde el enemigo explora nuevas acciones y explota el conocimiento adquirido en la Q-Table.

El algoritmo se implementa con la siguiente lógica:

1. Se evalúa el estado actual del enemigo y del jugador.
2. Se elige una acción basada en la Q-Table.
3. Se ejecuta la acción y se observa el nuevo estado.
4. Se calcula una recompensa en función del resultado de la acción.
5. Se actualiza la Q-Table utilizando la fórmula de Q-Learning.
6. Se repite el proceso en cada iteración del ciclo de juego.

Las recompensas y penalizaciones están diseñadas para incentivar un comportamiento inteligente. Por ejemplo:

- **Recompensa positiva (+100)** si el enemigo golpea al jugador con un ataque efectivo.
- **Recompensa menor (+500)** si logra acercarse al jugador.
- **Penalización (-50)** si el jugador está a la espalda del enemigo.
- **Penalización (-10)** si el enemigo no tiene visión del jugador.
- **Penalización (-5)** si el enemigo no puede ver al jugador con sus sensores.

Almacenamiento y Carga de la Q-Table

Para permitir que el enemigo aprenda de manera persistente, la Q-Table se almacena en un archivo JSON local. Esto garantiza que el conocimiento adquirido no se pierda al reiniciar la partida.

El sistema de almacenamiento se gestiona con las funciones:

- **save_q_table()**: Guarda la tabla en un archivo JSON.
- **load_q_table()**: Recupera la tabla desde el archivo al iniciar la partida.

Este mecanismo permite que el enemigo mejore su desempeño con cada sesión de juego y refine su comportamiento según el estilo del jugador.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones del proyecto

El desarrollo de este prototipo de videojuego ha sido un proceso complejo que ha requerido una cuidadosa gestión de recursos, adaptación a limitaciones técnicas y toma de decisiones estratégicas en torno a la implementación de mecánicas e inteligencia artificial. Uno de los principales desafíos encontrados fue la dificultad en la obtención de recursos gráficos y de sonido que se ajustaran a una estética coherente. Inicialmente, se probaron múltiples estilos visuales, pero la falta de uniformidad en los assets disponibles llevó a la necesidad de redefinir la dirección artística del proyecto, optando por una ambientación inspirada en el Japón antiguo y fantasioso. Este proceso implicó un esfuerzo adicional en la búsqueda y selección de recursos, descartando varios elementos que, aunque funcionales, no lograban integrarse adecuadamente en la propuesta final.

En cuanto a la implementación de inteligencia artificial, se exploraron distintas estrategias para dotar a los enemigos de comportamientos progresivos, diferenciando entre niveles de IA básica, media, alta y un nivel de IA basado en refuerzo. El desarrollo de la IA por refuerzo se convirtió en uno de los aspectos más interesantes del proyecto, ya que, debido a las limitaciones de documentación y herramientas en el motor *Godot*, fue necesario diseñar un algoritmo personalizado sin recurrir a *deep reinforcement learning*. La experimentación y ajuste de este sistema requirieron múltiples iteraciones, pero los resultados obtenidos demostraron que una versión simplificada de aprendizaje por refuerzo puede implementarse con éxito en entornos pequeños de videojuegos sin depender de herramientas externas avanzadas.

Otro aspecto clave en el desarrollo fue la gestión del tiempo y la priorización de tareas. Dado que el proyecto fue llevado a cabo de manera individual, fue necesario enfocarse en la funcionalidad esencial del prototipo. El cronograma se diseñó con base en ciclos de desarrollo ágiles, estableciendo *sprints* bien definidos, lo cual ayudó a manejar los imprevistos surgidos en el proceso.

Finalmente, este proyecto demuestra que, a pesar de las dificultades en la integración de múltiples recursos y en la implementación de una IA personalizada, es posible construir un prototipo funcional que represente la base de un videojuego más complejo. La documentación

generada y la experiencia adquirida sirven como referencia para futuras iteraciones, optimizaciones y expansiones del concepto inicial.

Recomendaciones para futuras investigaciones

A partir de la experiencia adquirida en este desarrollo, se pueden establecer varias recomendaciones para quienes deseen profundizar en la implementación de IA en videojuegos y en la gestión de recursos para proyectos independientes:

Definir desde el inicio una dirección artística clara y consistente. Una de las mayores dificultades del proyecto fue la búsqueda de recursos que se integraran visualmente de manera coherente. En futuras investigaciones, se recomienda establecer una línea gráfica desde las etapas iniciales para evitar la pérdida de tiempo en la selección y descarte de elementos incompatibles.

Explorar alternativas para mejorar la implementación de IA en Godot. Aunque la IA por refuerzo utilizada en este proyecto logró cumplir con su función, el motor aún presenta limitaciones en cuanto a documentación y herramientas especializadas para *deep reinforcement learning*. Se recomienda investigar librerías externas compatibles o considerar la combinación de *Godot* con otros lenguajes como *Python* para ampliar las posibilidades de aprendizaje automático en videojuegos.

Realizar pruebas controladas para evaluar la IA. Aunque este proyecto no contempló pruebas con usuarios externos, en futuras iteraciones sería útil diseñar experimentos que permitan analizar el desempeño de la IA de manera más objetiva. Esto ayudaría a identificar posibles mejoras en los algoritmos de toma de decisiones y en la adaptación del enemigo a distintos estilos de juego.

No se debe descartar el explorar la integración de IA en motores de juego más avanzados. Si bien Godot es un motor versátil, libre y accesible, algunos de sus sistemas de IA aún no están tan desarrollados como los de otros motores como Unity. Futuras investigaciones podrían evaluar la viabilidad de implementar técnicas de aprendizaje automático más avanzadas en entornos con mayor soporte y documentación.

ANEXO 1

GLOSARIO DE TÉRMINOS

A

- **AI (Inteligencia Artificial)**: Algoritmos y sistemas diseñados para simular comportamientos humanos en personajes no jugables (NPCs) o enemigos.
- **Assets**: Elementos gráficos, sonoros y de código utilizados en un videojuego, como texturas, modelos 3D, sonidos y animaciones.
- **Aseprite**: Software de edición de gráficos pixel art y animación, ampliamente utilizado para la creación de sprites en videojuegos 2D.

B

- **Build**: Versión compilada de un juego lista para pruebas o distribución.

C

- **Collider**: Elemento de física en un motor de videojuegos que define la colisión de un objeto.

E

- **Engine (Motor de Videojuegos)**: Software que facilita el desarrollo de videojuegos proporcionando herramientas para gráficos, física, IA y más (ej. Godot 4).

F

- **FPS (Cuadros por Segundo)**: Medida de la fluidez del juego.

G

- **Godot** : Motor de videojuegos de código abierto que permite desarrollar juegos en 2D y 3D con soporte para GDScript, C# y otros lenguajes.

H

- **Hitbox**: Zona invisible de colisión utilizada en personajes y objetos.
- **HUD (Heads-Up Display)**: Elementos visuales en pantalla que muestran información al jugador (vida, puntuación, mapa, etc.).

I

- **Import**: Proceso de traer assets externos al motor de videojuegos.
- **Instance (Instancia)**: Copia de un objeto en el juego sin necesidad de duplicar recursos.

N

- **NPC (Non-Playable Character)**: Personaje controlado por IA y no por un jugador.
- **Nodo**: Elemento fundamental en la estructura de Godot 4, utilizado para construir la jerarquía y lógica del juego.

P

- **Parallax**: Efecto visual donde objetos de fondo se mueven a diferentes velocidades para crear profundidad.

S

- **Sprite**: Imagen bidimensional utilizada en juegos 2D.
- **Sprite Sheet**: Imagen que contiene múltiples sprites organizados en una sola hoja para optimizar la animación y el rendimiento en videojuegos 2D.
- **Script**: Código utilizado para programar interacciones y lógica en un juego.

U

- **UI (User Interface)**: Interfaz de usuario en un videojuego que muestra menús y opciones.

W

- **World Building**: Creación del mundo y ambientación en un videojuego.

ANEXO 2

SPRINT BACKLOG

Sprint 1 Backlog

Tarea	**Descripción**	**Prioridad**	**Responsable**
Configurar el entorno de desarrollo	Instalar Godot Engine y preparar el entorno	Alta	Equipo
Implementar movimiento del personaje	Programar movimiento básico: correr, saltar	Alta	Equipo
Diseñar barra de vida funcional	Mostrar barra de salud en la interfaz	Alta	Equipo

Sprint 2 Backlog

Tarea	Descripción	Prioridad	Responsable
Crear el mapa del nivel inicial	Utilizar TileMaps para diseñar el nivel inicial	Alta	Equipo
Diseñar plataformas y obstáculos	Crear elementos básicos como plataformas y trampas	Alta	Equipo
Configurar física del juego	Establecer colisiones y gravedad	Alta	Equipo

Sprint 3 Backlog

Tarea	Descripción	Prioridad	Responsable
Crear enemigos con comportamientos básicos	Implementar IA para patrullaje	Alta	Equipo
Implementar ataques simples	Programar ataques básicos para enemigos y jugador	Alta	Equipo
Añadir interacción entre enemigos y jugador	Colisiones y reducción de salud en interacción	Alta	Equipo

Sprint 4 Backlog

Tarea	Descripción	Prioridad	Responsable

Integrar música de fondo y efectos de sonido	Añadir recursos de audio	Media	Equipo
Añadir animaciones a los personajes	Crear animaciones para movimiento y ataques	Alta	Equipo
Realizar pruebas funcionales	Detectar y corregir errores	Alta	Equipo

Sprint 5 Backlog

Tarea	Descripción	Prioridad	Responsable
Búsqueda de recursos gráficos	Investigar y recopilar assets compatibles con la ambientación	Alta	Equipo
Búsqueda de recursos de sonido	Buscar efectos de sonido y música adecuados	Alta	Equipo
Búsqueda de elementos de UI y menús	Encontrar una interfaz acorde a la ambientación	Alta	Equipo
Organización y filtrado de recursos	Clasificar y descartar elementos incompatibles	Alta	Equipo

Sprint 6 Backlog

Tarea	Descripción	Prioridad	Responsable
Implementación de escenarios y fondos	Adaptar y ajustar los escenarios con los recursos seleccionados	Alta	Equipo
Integración de UI y menús	Implementar los elementos de interfaz definitivos	Alta	Equipo
Implementación de sonidos y música	Integrar los efectos de sonido y la música seleccionada	Alta	Equipo
Ajuste de compatibilidad visual	Revisar que todos los elementos mantengan coherencia estilística	Alta	Equipo

Sprint 7 Backlog

Tarea	Descripción	Prioridad	Responsable
Desarrollo de la IA por refuerzo	Implementar algoritmo de aprendizaje por refuerzo simple	Alta	Equipo
Ajuste y refinamiento de la IA por refuerzo	Mejorar el comportamiento del enemigo final	Alta	Equipo
Modificación de escenarios para IA	Diseñar áreas donde el enemigo pueda detectar al jugador	Alta	Equipo
Pruebas internas con la IA	Observar el desempeño y ajustar parámetros	Alta	Equipo

Sprint 8 Backlog

Tarea	Descripción	Prioridad	Responsable
Refinamiento del enemigo final	Ajustar detalles del diseño y comportamiento	Alta	Equipo
Balanceo de dificultad	Ajustar la dificultad progresiva con base en observaciones	Alta	Equipo
Ajuste de animaciones y efectos visuales	Mejorar transiciones y fluidez de los elementos	Alta	Equipo
Últimos retoques de escenarios y UI	Pulir detalles gráficos y funcionales	Alta	Equipo

Sprint 9 Backlog

Tarea	Descripción	Prioridad	Responsable
Últimos ajustes y depuración	Corregir errores y optimizar la experiencia general	Alta	Equipo
Documentación interna del proyecto	Redactar información sobre IA, recursos y diseño	Alta	Equipo
Organización de archivos y versiones	Depurar archivos innecesarios y consolidar la versión final	Alta	Equipo
Presentación del prototipo	Preparar el prototipo para mostrar sus avances	Alta	Equipo

Sprint 10 - Backlog

Tarea	Descripción	Prioridad	Responsable
Revisión final del documento	Revisar la estructura, coherencia y redacción del documento final.	Alta	Equipo
Completar secciones pendientes	Redactar y finalizar cualquier apartado incompleto.	Alta	Equipo
Verificación de referencias y bibliografía	Asegurar que todas las referencias sean correctas y estén bien citadas.	Alta	Equipo
Revisión de formato y normas	Ajustar el documento a las normas de presentación establecidas.	Alta	Equipo
Revisión ortográfica y gramatical	Corregir errores de redacción y mejorar la claridad del contenido.	Alta	Equipo
Preparación de la presentación final	Sintetizar los puntos clave del documento para su exposición.	Alta	Equipo

DAILY SCRUM

SPRINT 1 - DAILY SCRUM

Día 1:

- **Logrado:** Instalación de Godot Engine y configuración básica del entorno.
- **Por hacer:** Implementar movimiento del personaje.
- **Obstáculos:** Ninguno.

Día 2:

- **Logrado:** Comenzó la implementación del movimiento del personaje (correr y saltar).
- **Por hacer:** Continuar con la programación del movimiento.
- **Obstáculos:** Ninguno.

Día 3:

- **Logrado:** Implementado el movimiento básico del personaje.
- **Por hacer:** Diseñar una barra de vida funcional.
- **Obstáculos:** No se encontraron imágenes adecuadas para la barra de vida; diseñarla manualmente toma tiempo.

Día 4:

- **Logrado:** Avances en la creación de la barra de vida.
- **Por hacer:** Completar la barra de vida con un diseño funcional.
- **Obstáculos:** Falta de recursos gráficos para la barra.

Día 5:

- **Logrado:** Avances en el movimiento del personaje.
- **Por hacer:** Revisión de tareas completadas en el sprint.
- **Obstáculos:** Pendiente la barra de vida, se lo dejará para sprints futuros.

SPRINT 2 - DAILY SCRUM

Día 1:

- **Logrado:** Inicio del diseño del nivel inicial con TileMaps.
- **Por hacer:** Continuar el diseño de plataformas y obstáculos.
- **Obstáculos:** Ninguno.

Día 2:

- **Logrado:** Diseño básico del nivel inicial completo.

- **Por hacer:** Configurar la física del juego (gravedad y colisiones).
- **Obstáculos:** Ninguno.

Día 3:

- **Logrado:** Física del juego implementada, colisiones funcionales.
- **Por hacer:** Refinar los elementos del nivel.
- **Obstáculos:** Ninguno.

Día 4:

- **Logrado:** Revisión de elementos del nivel.
- **Por hacer:** Finalizar pruebas funcionales del nivel.
- **Obstáculos:** Ninguno.

Día 5:

- **Logrado:** Ajustes finales en el nivel inicial.
- **Por hacer:** Validar tareas completas del sprint.
- **Obstáculos:** Ninguno.

Sprint 3 - Daily Scrum

Día 1:

- **Logrado:** Inicio de búsqueda de sprites para enemigos.
- **Por hacer:** Implementar IA básica para patrullaje, implementar máquina de estados modular.
- **Obstáculos:** Falta de sprites adecuados para los enemigos.

Día 2:

- **Logrado:** Avances en la IA de patrullaje.
- **Por hacer:** Configurar interacción enemigos-jugador, máquina de estados modular.
- **Obstáculos:** Colisiones funcionales, pero falta sistema de daño. Implementar la máquina de estados modular para el persona principal está complicando el proyecto

Día 3:

- **Logrado:** Continuación en la lógica de patrullaje.
- **Por hacer:** Implementar el daño en interacciones, implementación de máquina de estados modular.
- **Obstáculos:** Aún no hay sprites completos de enemigos. Aun no se logra implementar la máquina de estados modular.

Día 4:

- **Logrado:** Progresos menores en la IA.
- **Por hacer:** Revisión de la funcionalidad parcial, máquina de estados modular no funcional.
- **Obstáculos:** Tareas pendientes relacionadas con sprites limitan avances. Aun no se logra implementar la máquina de estados modular.

Día 5:

- **Logrado:** Validación parcial de la IA implementada. Se cancela la implementación de la máquina de estados modular, está consumiendo mucho tiempo y esfuerzo.
- **Por hacer:** Avanzar con las interacciones pendientes del sprint.
- **Obstáculos:** Falta de recursos gráficos.

Sprint 4 - Daily Scrum

Día 1:

- **Logrado:** Integración inicial de animaciones.
- **Por hacer:** Comenzar la integración de efectos de sonido.
- **Obstáculos:** Proyecto no listo para incorporar sonido; pendientes personajes.

Día 2:

- **Logrado:** Validación de animaciones funcionales.
- **Por hacer:** Continuar pruebas funcionales.
- **Obstáculos:** Sin avances en los personajes faltantes.

Día 3:

- **Logrado:** Pruebas iniciales de jugabilidad.
- **Por hacer:** Búsqueda de sprites.
- **Obstáculos:** Sin sprites avances en los personajes faltantes.

Día 4:

- **Logrado:** Revisión de progreso total.
- **Por hacer:** Búsqueda de sprites.
- **Obstáculos:** Recursos incompletos limitan rendimiento.

Día 5:

- **Logrado:** Validación de tareas funcionales del sprint.
- **Por hacer:** Tareas pendientes reprogramadas para futuros ciclos.

- **Obstáculos:** Falta de sprites y recursos necesarios.

Sprint 5 - Daily Scrum

Día 1:

- **Logrado:** Inicio de la búsqueda de recursos gráficos.
- **Por hacer:** Seguir investigando en diferentes plataformas de assets.
- **Obstáculos:** Muchos recursos son incompatibles con la estética deseada.

Día 2:

- **Logrado:** Evaluación de diferentes assets de personajes y escenarios.
- **Por hacer:** Filtrar y organizar los mejores recursos encontrados.
- **Obstáculos:** Variabilidad en los estilos artísticos dificulta la selección.

Día 3:

- **Logrado:** Avances en la búsqueda de efectos de sonido y música.
- **Por hacer:** Probar diferentes opciones y descartar las que no encajan.
- **Obstáculos:** Difícil encontrar sonidos que coincidan con la ambientación.

Día 4:

- **Logrado:** Evaluación de elementos de UI y menús disponibles.
- **Por hacer:** Probar combinaciones de menús y botones.
- **Obstáculos:** Algunos recursos de UI no son adaptables al juego.

Día 5:

- **Logrado:** Organización final de los recursos seleccionados.
- **Por hacer:** Prepararse para implementar los recursos en el juego.
- **Obstáculos:** Falta de cohesión total entre los elementos, aún hay ajustes por hacer.

Sprint 6 - Daily Scrum

Día 1:

- **Logrado:** Implementación inicial de los fondos y escenarios.
- **Por hacer:** Ajustar detalles en el mapa para una mejor coherencia visual.
- **Obstáculos:** Algunos assets no tienen la resolución adecuada.

Día 2:

- **Logrado:** Integración inicial de la UI en el juego.
- **Por hacer:** Ajustar fuentes, botones y tamaño de los elementos.

- **Obstáculos:** Algunas interfaces necesitan rediseño para ser funcionales.

Día 3:

- **Logrado:** Implementación básica de efectos de sonido en acciones clave.
- **Por hacer:** Revisar volumen y transiciones entre sonidos.
- **Obstáculos:** Falta de variedad en algunos efectos de sonido.

Día 4:

- **Logrado:** Integración de la música en los diferentes escenarios.
- **Por hacer:** Ajustar el volumen y transición de las pistas.
- **Obstáculos:** Algunas pistas no encajan bien con la ambientación.

Día 5:

- **Logrado:** Ajuste final de la coherencia visual entre UI y escenarios.
- **Por hacer:** Refinar detalles menores en la UI y elementos interactivos.
- **Obstáculos:** Pequeñas inconsistencias entre algunos elementos visuales.

Sprint 7 - Daily Scrum

Día 1:

- **Logrado:** Inicio de la implementación de la IA por refuerzo.
- **Por hacer:** Definir estados y recompensas iniciales.
- **Obstáculos:** Poca documentación sobre implementación en Godot.

Día 2:

- **Logrado:** Ajuste de parámetros iniciales para el entrenamiento del enemigo.
- **Por hacer:** Evaluar si la IA responde adecuadamente a estímulos.
- **Obstáculos:** El aprendizaje es más lento de lo esperado.

Día 3:

- **Logrado:** Ajustes en las recompensas y castigos de la IA.
- **Por hacer:** Implementar zonas de detección del jugador.
- **Obstáculos:** Algunos valores de recompensa generan comportamientos extraños.

Día 4:

- **Logrado:** Refinamiento de las áreas donde el enemigo detecta al jugador.
- **Por hacer:** Optimizar el tiempo de reacción del enemigo.
- **Obstáculos:** Aún hay situaciones donde el enemigo se comporta de manera errática.

Día 5:

- **Logrado:** IA responde mejor en ciertas situaciones.
- **Por hacer:** Ajustar pequeños detalles de su comportamiento.
- **Obstáculos:** Algunos errores de detección aún persisten.

Sprint 8 - Daily Scrum

Día 1:

- **Logrado:** Mejoras en la animación del enemigo final.
- **Por hacer:** Revisar los detalles de la animación en combate.
- **Obstáculos:** Algunas animaciones no están sincronizadas.

Día 2:

- **Logrado:** Ajuste de la dificultad de los enemigos en el juego.
- **Por hacer:** Revisar si el equilibrio en la jugabilidad es adecuado.
- **Obstáculos:** Algunos enemigos son demasiado predecibles.

Día 3:

- **Logrado:** Implementación de nuevas transiciones visuales y efectos.
- **Por hacer:** Revisar tiempos de respuesta y sincronización de animaciones.
- **Obstáculos:** Algunas animaciones se ven desincronizadas.

Día 4:

- **Logrado:** Últimos retoques en escenarios y detalles gráficos.
- **Por hacer:** Revisar si hay inconsistencias visuales.
- **Obstáculos:** Pequeñas diferencias de estilos aún visibles.

Día 5:

- **Logrado:** Refinamiento final de la experiencia visual del juego.
- **Por hacer:** Hacer ajustes menores en algunos sprites.
- **Obstáculos:** Aún se detectan pequeñas inconsistencias en algunos detalles.

Sprint 9 - Daily Scrum (Entrega del Demo)

Día 1:

- **Logrado:** Organización y depuración de archivos del proyecto.

- **Por hacer:** Verificar que todos los elementos del juego (escenarios, enemigos, UI, IA) estén funcionando correctamente.
- **Obstáculos:** Algunos archivos duplicados generan confusión en la versión final.

Día 2:

- **Logrado:** Correcciones finales en la IA del enemigo final.
- **Por hacer:** Revisar que la IA responda correctamente en todas las áreas del juego.
- **Obstáculos:** La IA básica noback dragon tiene problemas con la detección del jugador en ciertas zonas del mapa.

Día 3:

- **Logrado:** Últimos ajustes en los escenarios y enemigos.
- **Por hacer:** Asegurar que no haya errores gráficos o problemas en la jugabilidad.
- **Obstáculos:** Algunos sprites y animaciones no encajan perfectamente en ciertas situaciones.

Día 4:

- **Logrado:** Versión final del demo.
- **Por hacer:** Verificar que el demo funcione correctamente en la versión de escritorio.
- **Obstáculos:** -.

Día 5:

- **Logrado:** Verificación final del demo.
- **Por hacer:** -.
- **Obstáculos:** -.

Sprint 10 - Daily Scrum

Día 1:

- **Logrado:** Revisión inicial del documento, detección de secciones incompletas.
- **Por hacer:** Completar las partes faltantes y mejorar la cohesión entre secciones.
- **Obstáculos:** Algunos apartados requieren mayor desarrollo teórico.

Día 2:

- **Logrado:** Avance en la redacción de secciones pendientes.
- **Por hacer:** Revisar la coherencia y conexión entre apartados.

- **Obstáculos:** Diferencias en el nivel de detalle entre secciones.

Día 3:

- **Logrado:** Verificación y corrección de referencias bibliográficas.
- **Por hacer:** Ajustar formato y normas de presentación.
- **Obstáculos:** Necesidad de estandarizar el estilo de citación.

Día 4:

- **Logrado:** Corrección de errores ortográficos y gramaticales.
- **Por hacer:** Última revisión para garantizar la calidad del documento.
- **Obstáculos:** Se identifican inconsistencias menores en la redacción.

Día 5:

- **Logrado:** Preparación final del documento y ajustes en la presentación.
- **Por hacer:** Entrega del documento y revisión general del proceso.
- **Obstáculos:** -

RETROSPECTIVA DE SPRINTS

Revisión del Sprint 1

Logros:

- Se instaló y configuró correctamente **Godot Engine**.
- Se implementó el **movimiento básico** del personaje, incluyendo correr y saltar.
- Se avanzó en la creación de la **barra de vida funcional (el apartado gráfico se lo aplazará hasta un sprint futuro donde se integren recursos gráficos)**.

Pendientes:

- La barra de vida no se completó en su totalidad debido a la falta de imágenes adecuadas.

Obstáculos:

- La falta de recursos gráficos adecuados complicó la implementación de la barra de vida.

Retrospectiva del Sprint 1

Aspectos positivos:

- Se logró implementar la base del movimiento del personaje sin problemas técnicos.
- No hubo dificultades técnicas significativas en la configuración del entorno de desarrollo.

Áreas de mejora:

- Se requiere una mejor estrategia para la adquisición o generación de recursos gráficos, evitando retrasos en tareas dependientes.

Acciones para el próximo sprint:

- Incluir en la planificación la identificación de posibles requerimientos gráficos antes de asignar tareas que dependan de estos recursos.

Revisión del Sprint 2

Logros:

- Se diseñó el nivel inicial utilizando **TileMaps**.
- Se crearon plataformas y obstáculos básicos.

- Se configuró la física del juego, implementando colisiones y gravedad.
- Se realizaron ajustes y revisiones finales en el nivel.

Pendientes:

- No se identificaron tareas pendientes, ya que todas las actividades planificadas para el sprint fueron completadas.

Obstáculos:

- No se reportaron obstáculos a lo largo del sprint.

Retrospectiva del Sprint 2

Aspectos positivos:

- Se logró completar todas las tareas del sprint dentro del tiempo estimado.
- No se presentaron bloqueos técnicos, lo que permitió un avance continuo.
- La planificación del sprint fue efectiva, asegurando que todas las tareas estuvieran alineadas con los objetivos.

Áreas de mejora:

- Se recomienda definir criterios de calidad más detallados para evaluar la jugabilidad del nivel desde etapas tempranas.

Acciones para el próximo sprint:

- Estar más atento a detectar posibles mejoras en la jugabilidad.

Revisión del Sprint 3

Logros:

- Se implementó parcialmente la inteligencia artificial (IA) de patrullaje para los enemigos.
- Se avanzó en la configuración de interacciones entre los enemigos y el jugador.
- Se realizaron validaciones parciales sobre la IA implementada.

Pendientes:

- No se completó la implementación de la máquina de estados modular, por lo que se decidió cancelarla debido al alto consumo de tiempo y esfuerzo.
- No se logró obtener los sprites adecuados para los enemigos, lo que limitó el progreso en la interacción visual y funcional de estos con el jugador.
- La implementación del sistema de daño entre el jugador y los enemigos no se finalizó.

Obstáculos:

- La falta de recursos gráficos adecuados ralentizó el desarrollo de la interacción entre enemigos y el jugador.
- La máquina de estados modular resultó ser demasiado compleja en esta fase del proyecto, lo que generó retrasos en otras tareas clave.

Retrospectiva del Sprint 3

Aspectos positivos:

- Se logró un progreso parcial en la implementación de la IA de patrullaje.
- Se identificó a tiempo la dificultad de la máquina de estados modular, lo que permitió tomar la decisión de cancelarla y reasignar esfuerzos.
- Se validaron las funcionalidades desarrolladas para asegurar su correcto funcionamiento dentro del juego.

Áreas de mejora:

- Es necesario planificar con mayor anticipación la disponibilidad de recursos gráficos para evitar bloqueos en tareas dependientes.
- Se recomienda pensar bien antes de comprometer tiempo significativo en la implementación de sistemas complejos como la máquina de estados modular.
- La gestión de prioridades debe ser más flexible para redistribuir esfuerzos en caso de bloqueos persistentes.

Acciones para el próximo sprint:

- Priorizar tareas que aseguren la jugabilidad sin depender de implementaciones avanzadas que puedan retrasar el progreso del proyecto.

Revisión del Sprint 4

Logros:

- Se integraron las animaciones iniciales para los personajes.
- Se realizaron pruebas funcionales preliminares de jugabilidad.

Pendientes:

- La integración de efectos de sonido no se completó debido a que el proyecto aún no estaba en condiciones óptimas para ello.
- Algunas tareas fueron reprogramadas para futuros ciclos de desarrollo.

Obstáculos:

- La ausencia de ciertos recursos gráficos limitó la implementación eficiente de las pruebas funcionales.
- El proyecto no estaba completamente preparado para la incorporación de efectos de sonido, lo que retrasó su implementación.

Retrospectiva del Sprint 4

Aspectos positivos:

- Se logró un avance significativo en las animaciones y pruebas de jugabilidad.
- Se identificaron áreas críticas que necesitan más recursos para su correcta implementación.
- Se realizó una revisión general de los avances, lo que permitió reorganizar tareas para futuros sprints.

Áreas de mejora:

- Es necesario garantizar la disponibilidad de recursos gráficos antes de asignar tareas dependientes.

Acciones para el próximo sprint:

- Reforzar la gestión de recursos gráficos antes de iniciar el desarrollo de nuevas funcionalidades.

Revisión del Sprint 5

Logros:

- Se inició y avanzó en la búsqueda de recursos gráficos para personajes y escenarios.
- Se evaluaron diferentes opciones de efectos de sonido y música para la ambientación del juego.
- Se realizó una revisión y prueba de diversos elementos de UI y menús.
- Se organizó un conjunto final de recursos gráficos, de sonido y de interfaz para su implementación en el juego.

Pendientes:

- Aún no se ha logrado una cohesión total entre los recursos seleccionados, lo que requerirá ajustes adicionales en el próximo sprint.
- Se necesita una mayor alineación estética entre los diferentes elementos visuales y de interfaz.

Obstáculos:

- La compatibilidad de los recursos gráficos con la estética deseada fue un desafío recurrente.
- La variabilidad en los estilos artísticos de los assets dificultó la selección y filtrado de los más adecuados.
- Encontrar sonidos que coincidieran con la ambientación prevista resultó complicado, lo que generó demoras en la integración de efectos sonoros.
- Algunos recursos de UI no se adaptaban bien al diseño del juego, lo que requirió una exploración más exhaustiva.

Retrospectiva del Sprint 5

Aspectos positivos:

- Se realizó una selección detallada de recursos visuales y auditivos, lo que permitirá una implementación más ágil en los próximos sprints, además se ha decidido hacer un solo nivel de juego.
- Se logró una mejor comprensión de las necesidades estéticas y estilísticas del proyecto.

- Se organizó de manera efectiva el material seleccionado para facilitar su integración.

Áreas de mejora:

- Se recomienda establecer criterios de selección más claros desde el inicio para evitar la acumulación de tareas de ajuste en fases posteriores.
- Es necesario definir una dirección artística más concreta para mejorar la cohesión de los recursos elegidos.

Acciones para el próximo sprint:

- Aplicar los recursos seleccionados en el juego y realizar los ajustes necesarios para lograr una coherencia visual y auditiva adecuada.

Revisión del Sprint 6

Logros:

- Se implementaron fondos y escenarios iniciales en el juego.
- Se integraron elementos de UI y menús en su primera versión funcional.
- Se añadieron efectos de sonido básicos y se integró música de fondo en los escenarios.
- Se realizaron ajustes para mejorar la coherencia visual entre los elementos de UI y los escenarios.

Pendientes:

- Persisten pequeñas inconsistencias visuales entre los elementos gráficos que requieren ajustes adicionales.
- Es necesario rediseñar ciertas interfaces para mejorar su funcionalidad y adaptabilidad.

Obstáculos:

- Algunos assets no cumplen con la resolución esperada, lo que afectó la calidad visual en ciertas áreas.
- Falta de variedad en los efectos de sonido, lo que limitó la personalización de las acciones del jugador.
- Algunas pistas musicales no encajan perfectamente con el contexto del juego, lo que dificulta la inmersión total del usuario.

Retrospectiva del Sprint 6

Aspectos positivos:

- Se logró una integración funcional de escenarios, música y efectos de sonido, cumpliendo con los objetivos principales del sprint.
- Se mejoró considerablemente la coherencia visual entre los elementos del juego gracias a los ajustes realizados.
- La primera iteración de la UI ofrece una base sólida para futuros refinamientos.

Áreas de mejora:

- Se necesita establecer criterios más estrictos para la selección de recursos gráficos y sonoros antes de su integración.
- Mejorar la planificación de los ajustes de diseño para minimizar inconsistencias visuales y funcionales.

Acciones para el próximo sprint:

- Resolver inconsistencias gráficas restantes y asegurar la cohesión visual en todos los aspectos del diseño.
- Rediseñar elementos de UI problemáticos para garantizar una experiencia de usuario óptima.

Revisión del Sprint 7

Logros:

- Se inició la implementación de la IA por refuerzo, definiendo sus estados y recompensas iniciales.
- Se ajustaron los parámetros iniciales del entrenamiento del enemigo y su respuesta a estímulos.
- Se implementaron zonas de detección del jugador para mejorar la interacción con el entorno.
- Se realizaron ajustes en el sistema de recompensas y castigos, refinando el comportamiento de la IA.

- Se optimizaron las áreas donde el enemigo detecta al jugador, mejorando su tiempo de reacción.
- La IA mostró mejoras en su respuesta en ciertas situaciones.

Pendientes:

- Persisten algunos errores en la detección del jugador que requieren ajustes adicionales.
- Se deben corregir ciertos comportamientos erráticos que surgen en situaciones específicas.

Obstáculos:

- La documentación sobre la implementación de aprendizaje por refuerzo en **Godot** es limitada, lo que ralentizó el desarrollo.
- Algunos valores de recompensa generaron comportamientos inesperados en la IA.
- El aprendizaje de la IA es más lento de lo previsto, lo que ha requerido ajustes constantes.
- Persisten ciertos errores en la detección del jugador, lo que afecta la jugabilidad.

Retrospectiva del Sprint 7

Aspectos positivos:

- Se logró una integración inicial funcional de la IA dentro del juego.
- Se realizaron pruebas y ajustes en los parámetros de recompensa y detección del jugador, lo que permitió mejorar la respuesta del enemigo.
- Se establecieron bases sólidas para la optimización futura del comportamiento de la IA.

Áreas de mejora:

- Es necesario profundizar en la documentación y buscar fuentes externas para mejorar la implementación del aprendizaje por refuerzo en **Godot**.
- Se debe evaluar la posibilidad de simplificar ciertos aspectos del entrenamiento de la IA para acelerar su aprendizaje sin comprometer su funcionalidad.

Acciones para el próximo sprint:

- Refinar los valores de recompensa y castigo para evitar comportamientos erráticos.

Revisión del Sprint 8

Logros:

- Se mejoraron las animaciones del enemigo final, incluyendo ajustes en combate.
- Se ajustó la dificultad progresiva de los enemigos, optimizando el balance en la jugabilidad.
- Se implementaron nuevas transiciones visuales y efectos para mejorar la fluidez de la experiencia.
- Se realizaron retoques finales en escenarios y detalles gráficos.

Pendientes:

- Persisten pequeñas inconsistencias visuales entre algunos elementos gráficos.
- Se necesitan ajustes menores en ciertos sprites para lograr una mayor armonía visual.
- Algunas animaciones presentan desincronización y requieren revisión adicional.

Obstáculos:

- La falta de recursos adecuados para animaciones de transición dificultó la mejora de la fluidez visual.
- Algunos enemigos aún presentan comportamientos predecibles, lo que afecta la variabilidad en la dificultad del juego.
- Existen diferencias visuales entre algunos elementos gráficos, lo que afecta la coherencia visual general.

Retrospectiva del Sprint 8

Aspectos positivos:

- Se lograron mejoras significativas en la calidad visual y en la jugabilidad del juego.
- Se implementaron transiciones más fluidas, mejorando la experiencia del usuario.
- El balance de dificultad se ajustó de acuerdo con pruebas y observaciones, logrando una progresión más equilibrada.

Áreas de mejora:

- Es necesario optimizar el comportamiento de los enemigos para evitar patrones predecibles en combate.
- Se recomienda una revisión detallada de la sincronización de animaciones para evitar desajustes en la experiencia visual.

Acciones para el próximo sprint:

- Realizar una fase final de ajustes en sprites y transiciones para garantizar coherencia visual.
- Refinar los patrones de comportamiento de los enemigos para mejorar su imprevisibilidad y desafío.

Revisión del Sprint 9

Logros:

- Se realizó la organización y depuración de archivos, consolidando la versión final del proyecto.
- Se implementaron correcciones finales en la **IA del enemigo final**, mejorando su respuesta en la jugabilidad.
- Se ajustaron los **escenarios y enemigos**, optimizando su integración visual y funcional.
- Se completó la **versión final del demo**, listo para pruebas y validaciones.
- Se verificó el correcto funcionamiento del demo en la versión de escritorio.

Pendientes:

- Persisten pequeñas inconsistencias en algunos sprites y animaciones en ciertas situaciones.
- Se detectaron problemas de visualización en diferentes resoluciones que podrían requerir ajustes adicionales.
- Algunas áreas del mapa aún presentan **fallos en la detección del jugador por parte de la IA**.

Obstáculos:

- La existencia de archivos duplicados generó confusión en la organización de la versión final.
- Hubo dificultades en la sincronización de elementos visuales en **diferentes resoluciones de pantalla**.
- Se identificaron errores menores en la **IA en ciertas zonas del mapa**, afectando su capacidad de detección.

Retrospectiva del Sprint 9

Aspectos positivos:

- Se logró consolidar una versión jugable del demo con todos los elementos principales integrados.
- La depuración del código y archivos facilitó la organización final del proyecto.
- Se optimizó la IA del enemigo final, mejorando su comportamiento en la mayoría de las situaciones.

Áreas de mejora:

- La gestión de archivos debe ser más rigurosa en futuros proyectos para evitar duplicaciones innecesarias.

Acciones para la presentación del prototipo:

- Documentar los ajustes realizados en la IA para facilitar futuras optimizaciones.
- Preparar material de apoyo para la presentación del demo, incluyendo **una lista de características implementadas y posibles mejoras a futuro**.

Revisión del Sprint 10

Logros:

- Se completaron todas las secciones del documento con la profundidad requerida.
- Se corrigieron errores de redacción, formato y referencias bibliográficas.
- Se garantizó la coherencia y cohesión entre los distintos apartados.
- Se ajustó el documento a las normas de presentación establecidas.
- Se preparó la presentación final basada en los resultados del trabajo.

Pendientes:

- -.

Obstáculos:

- Se requirió un esfuerzo adicional para ajustar el formato de todo el documento.
- Hubo que realizar ajustes de último momento en las referencias y el formato.

Retrospectiva del Sprint 10

Aspectos positivos:

- Se realizaron múltiples revisiones que mejoraron la claridad y calidad del contenido.

Áreas de mejora:

- Se recomienda distribuir mejor la carga de trabajo en la redacción de futuras investigaciones para evitar concentrar tareas en los últimos sprints.
- Es importante establecer criterios uniformes de redacción desde el inicio para evitar inconsistencias en la redacción.
- La gestión de referencias bibliográficas debe abordarse con más antelación para evitar correcciones de última hora.

Acciones para el futuro:

- Implementar revisiones parciales en cada sprint para evitar acumulación de tareas en la fase final.
- Definir desde el inicio un estilo de citación y formato uniforme.
- Documentar el proceso de desarrollo del documento para futuras investigaciones.

ANEXO 3

Glosario de Nodos en Godot

A

- **AnimatedSprite2D**: Nodo que reproduce animaciones de sprites desde una hoja de sprites o múltiples imágenes, facilitando la gestión de animaciones sin necesidad de un **AnimationPlayer**.
- **AnimationPlayer**: Nodo que maneja la reproducción de animaciones para modificar propiedades de otros nodos en la escena.
- **Area2D**: Nodo que detecta la presencia de otros cuerpos dentro de un área específica, útil para colisiones y detección de eventos.
- **AudioStreamPlayer2D**: Nodo que permite reproducir audio en un entorno 2D, ideal para efectos de sonido y música.

C

- **Camera2D**: Nodo de cámara que permite seguir a personajes u objetos, ajustando la vista en el espacio 2D.
- **CanvasLayer**: Nodo que maneja elementos de interfaz de usuario y superposiciones sin ser afectado por el movimiento de la cámara.
- **CollisionShape2D**: Nodo que define la forma de colisión de un objeto en 2D, utilizando formas geométricas como rectángulos o círculos.

M

- **Marker2D**: Nodo que actúa como un punto de referencia en el espacio 2D, útil para posiciones predefinidas.

N

- **Node2D**: Nodo base para todos los nodos 2D, maneja posición, rotación y escala.

P

- **ParallaxBackground**: Nodo que permite crear un efecto de paralaje en fondos 2D, mejorando la sensación de profundidad.

- **Path2D**: Nodo que define una ruta en el espacio 2D, útil para movimiento basado en curvas o trayectorias.

R

- **RayCast2D**: Nodo que detecta objetos en una línea recta, utilizado para colisiones, detección de enemigos o disparos.

S

- **Sprite2D**: Nodo que representa imágenes en 2D, ideal para personajes, objetos y decoraciones.

T

- **TileMap**: Nodo que gestiona mosaicos en una cuadrícula, facilitando la creación de niveles basados en tiles.

ANEXO 4

Algoritmos del Demo

ALGORITMO DEL JUGADOR

Algoritmo de jugador (ENZO)

```
### -- VARIABLES GLOBALES -- ###
INICIALIZAR:
    - **Estados disponibles**:
        - IDLE, RUN, JUMP, FALL, WALL_CLIMB, DASH
        - ATTACK_SIMPLE, DEFEND, THROW, DASH_ATTACK, CHARGE_ATTACK
    - **Referencias a nodos**:
        - sprite = Nodo Sprite2D (manejo de orientación)
        - animation_player = Nodo AnimationPlayer (manejo de animaciones)
        - raycast_left / raycast_right = RayCasts para detectar paredes
        - throw_point = Marcador de lanzamiento del Shuriken
    - **Sistema de combate**:
        - attack_area_uno = Área de colisión para ataque simple
        - attack_duration = 0.4 seg
        - attack_timer = 0.0 seg
        - damage = 0
    - **Sistema de movimiento**:
        - SPEED = 600.0
        - JUMP_VELOCITY = -1000.0
        - DASH_SPEED = 2000.0
        - DASH_DURATION = 0.3 seg
        - WALL_CLIMB_SPEED = 200.0
        - gravity = 500.0
    - **Dash y ataques**:
        - dash_timer = 0.0
        - dash_direction = 0
        - can_dash_attack = True (cooldown de 1.5 seg)
    - **Sistema de resistencia**:
        - stamina = 100.0
        - stamina_regen_rate = 10.0 / seg
        - stamina_depletion_charge_attack = 40.0
    - **Variables auxiliares**:
        - is_facing_right = True (orientación inicial)
        - health = 100 (vida del jugador)
        - coin_count = 0 (contador de monedas)
############-- PROCESO DE FÍSICA (_physics_process) --####################
FUNCIÓN _physics_process(delta):
```

```
    - **Manejo de Estados**:
        - MATCH state:
            - IDLE → `handle_idle_state(delta)`
            - RUN → `handle_run_state(delta)`
            - JUMP → `handle_jump_state(delta)`
            - FALL → `handle_fall_state(delta)`
            - WALL_CLIMB → `handle_wall_climb_state(delta)`
            - DASH → `handle_dash_state(delta)`
            - ATTACK_SIMPLE → `handle_attack_simple_state(delta)`
            - DEFEND → `handle_defend_state(delta)`
            - THROW → `handle_throw_state(delta)`
            - CHARGE_ATTACK → `handle_charge_attack_state(delta)`
            - DASH_ATTACK → `handle_dash_attack_state(delta)`
    - **Regeneración de resistencia**:
        - `stamina = min(100, stamina + stamina_regen_rate * delta)`
    - **Captura de inputs**:
        - SI `"ui_throw"` es presionado → `start_throw()`
        - **Aplicar movimiento** → `move_and_slide()`
#############################-- ESTADOS DE MOVIMIENTO --#############################
FUNCIÓN handle_idle_state(delta):
    - **Reiniciar variables**:
        - Restaurar velocidad de animaciones → `animation_player.speed_scale = 1.0`

    - **Transiciones**:
        - SI `"ui_dash_attack"` presionado **y** `can_dash_attack` → `start_charge_attack()`
        - SI `"ui_throw"` presionado → `start_throw()`
        - SI `"ui_defend"` presionado → `change_state(State.DEFEND)`
        - SI `"ui_attack_simple"` presionado → `start_attack_simple()`
        - SI jugador no está en el suelo → `change_state(State.FALL)`
        - SI jugador está tocando una pared **y** `"ui_select"` presionado → `change_state(State.WALL_CLIMB)`
        - SI `"ui_dash"` presionado → `start_dash()`
        - SI `"ui_accept"` presionado → `change_state(State.JUMP)`
        - SI `"ui_left"` o `"ui_right"` presionado → `change_state(State.RUN)`
    - **Reproducir animación** → `"idle"`
FUNCIÓN handle_run_state(delta):
    - **Verificar entrada de ataques**:
        - `"ui_dash_attack"` presionado → `start_charge_attack()`
        - `"ui_throw"` presionado → `start_throw()`
        - `"ui_defend"` presionado → `change_state(State.DEFEND)`
        - `"ui_attack_simple"` presionado → `start_attack_simple()`

    - **Verificar cambios de estado**:
        - SI no está en el suelo → `change_state(State.FALL)`
```

```
        - SI `"ui_select"` presionado **y** tocando pared → `change_state(State.WALL_CLIMB)`
        - SI `"ui_dash"` presionado → `start_dash()`
        - SI `"ui_left"` y `"ui_right"` no están presionados → `change_state(State.IDLE)`
    - **Actualizar movimiento**:
        - `velocity.x = dirección * SPEED`
        - **Voltear sprite según dirección** → `sprite.flip_h = dirección < 0`
FUNCIÓN handle_jump_state(delta):
    - **Aplicar control en el aire** → `apply_air_control(delta)`
    - **Verificar cambios de estado**:
        - `"ui_throw"` presionado → `start_throw()`
        - `"ui_defend"` presionado → `change_state(State.DEFEND)`
        - `"ui_dash"` presionado → `start_dash()`
        - `"ui_accept"` presionado **y** `jumps_left > 0`:
            - `velocity.y = JUMP_VELOCITY`
            - `jumps_left -= 1`
        - SI tocando pared **y** `"ui_select"` presionado → `change_state(State.WALL_CLIMB)`
        - SI `velocity.y > 0` → `change_state(State.FALL)`
    - **Aplicar gravedad**:
        - `velocity.y += gravity * delta`
#############################-- ESTADOS DE COMBATE --#############################
FUNCIÓN start_attack_simple():
    - **Guardar estado previo**: `state_before_attack = state`
    - **Iniciar temporizador**: `attack_timer = attack_duration`
    - **Cambiar de estado**: `change_state(State.ATTACK_SIMPLE)`
FUNCIÓN handle_attack_simple_state(delta):
    - **Reducir temporizador de ataque**:
        - SI `attack_timer <= 0` → `change_state(State.IDLE if is_on_floor() else State.FALL)`
    - **Ejecutar ataque** según estado previo:
        - IDLE → `animated_sprite.play("attack_1")`
        - RUN → `animated_sprite.play("attack_1")`
    - **Activar área de colisión** → `activate_collision_area("AttackArea_uno")`
    - **Aplicar daño** → `apply_damage("AttackArea_uno", 20)`
FUNCIÓN start_charge_attack():
    - SI `state != State.IDLE` **y** `state != State.RUN` → RETORNAR
    - **Reducir resistencia**: `stamina -= 40`
    - **Cambiar estado**: `change_state(State.CHARGE_ATTACK)`
    - **Reproducir animación**: `animated_sprite.play("charge_attack")`
FUNCIÓN handle_charge_attack_state(delta):
    - SI `"ui_dash_attack"` liberado:
        - `velocity.x = dirección * 600`
        - `start_dash_attack()`
    - **Bloquear movimiento** → `velocity.x = 0`
#############################-- SISTEMA DE DAÑO --#############################
```

```
FUNCIÓN take_damage(amount, knockback_vector):
    - SI `state == State.DEFEND`:
        - `"¡Ataque bloqueado!"`
        - `animated_sprite.play("defence_hit")`
        - RETORNAR
    - **Reducir vida** → `health -= amount`
    - **Efecto de parpadeo** → `flicker_effect()`
    - **Aplicar retroceso** → `velocity += knockback_vector`
    - **Eliminar jugador si la vida llega a 0** → `die()`
FUNCIÓN die():
    - `"El jugador ha muerto"`
##############################-- SISTEMA DE MONEDAS --##############################
FUNCIÓN add_coins(amount):
    - `coin_count += amount`
    - `"Monedas:", coin_count`
```

ALGORITMOS DE ENEMIGOS

Algoritmo IA BÁSICA Ranas

```
INICIALIZAR variables:
    speed = 100.0
    gravity = 500.0
    health = 30
    damage = 10
    direction = initial_direction
    is_active = True
    node_pool = buscar "NodePool" en la escena
    coin_scene = referencia a la escena de monedas
    coin_count = 5
FUNCIÓN _ready():
    SI node_pool ES NULL → MOSTRAR ERROR
    Reproducir animación "run"
LOOP _physics_process(delta):
    SI NOT is_active:
        Reproducir animación "idle"
        RETORNAR
    APLICAR gravedad a velocity.y
    MOVER en dirección actual → velocity.x = direction * speed
    Ejecutar move_and_slide()
    SI detecta colisión con pared:
        Cambiar dirección (direction *= -1)
        Voltear sprite (sprite.flip_h)
    SI está en el suelo:
        velocity.y = 0
FUNCIÓN take_damage(amount, knockback_vector):
    health -= amount
    MOSTRAR "El enemigo ha recibido daño"
    APLICAR knockback a velocity
    Ejecutar move_and_slide()
    Reproducir animación "hurt"
    ESPERAR animación terminada
    SI health <= 0 → LLAMAR die()
    SI NO → Reproducir animación "run"
FUNCIÓN die():
    Reproducir animación "death"
    DESACTIVAR colisiones
    ESPERAR animación terminada
    SI coin_scene EXISTE:
        GENERAR coin_count monedas en la posición del enemigo
```

```
        ASIGNAR jugador como objetivo de las monedas
    SI node_pool EXISTE:
        GUARDAR enemigo en node_pool
    SI NO → ELIMINAR enemigo (queue_free())
FUNCIÓN _on_area_2d_damage_body_entered(body):
    SI body pertenece a "player_group":
        MOSTRAR "El enemigo tocó al nodo"
        APLICAR knockback al jugador con dirección opuesta
        LLAMAR body.take_damage(damage, knockback_force)
FUNCIÓN respawn(new_position):
    SI node_pool EXISTE:
        RECUPERAR enemigo de node_pool y colocarlo en new_position
        ACTIVAR enemigo
        MOSTRAR "Enemigo reaparecido"
```

Algoritmo IA BÁSICA noback dragón

```
### -- VARIABLES GLOBALES -- ###
INICIALIZAR:
    - speed = 600.0  # Velocidad de ataque en picada
    - damage = 20  # Daño al impactar con el jugador
    - gravity = 500.0  # Gravedad aplicada al dragón
    - player = null  # Referencia al jugador
    - attacking = False  # Indica si el dragón está atacando
    - has_exploded = False  # Controla si ya explotó para evitar múltiples detonaciones
    - state = IDLE  # Estado inicial del dragón
    - direction = Vector2.ZERO  # Dirección de movimiento hacia el jugador
    - sound_explosion = cargar("res://assets/sound/explosion.wav")  # **(Nuevo)** Sonido de explosión
    - sound_dive = cargar("res://sonidos/dive_attack.wav")  # **(Nuevo)** Sonido al iniciar el ataque
    - node_pool = obtener "NodePool" de la escena  # **(Nuevo)** Manejo de Object Pooling
    - is_active = True  # Controla si el dragón está activo en la escena
#############################-- INICIALIZACIÓN (_ready) --#############################
FUNCIÓN _ready():
    - SI node_pool **NO existe**:
        - MOSTRAR ERROR: `"No se encontró NodePool en la escena"`
#############################-- PROCESO DE FÍSICA (_physics_process) --#############################
FUNCIÓN _physics_process(delta):
    - SI `not is_active`:
        - Reproducir animación `"idle"`
        - RETORNAR  # No ejecutar más lógica si está inactivo
    - **Lógica de la Máquina de Estados**:
        MATCH state:
            - IDLE → LLAMAR `_idle_state()`
            - DIVE → LLAMAR `_dive_state(delta)`
            - EXPLODE → LLAMAR `_explode_state()`
    - **Aplicar movimiento** con `move_and_slide()`
#############################-- MÁQUINA DE ESTADOS --#############################
FUNCIÓN _idle_state():
    - Reproducir animación `"idle"`
    - Establecer `velocity = Vector2.ZERO`
    - **Detectar si el jugador está en el área de patrullaje**:
        - SI `_detecta_jugador() == True`:
            - Cambiar `state = DIVE`
            - Establecer `attacking = True`
            - Reproducir animación `"attack_1"`
            - **Reproducir sonido de ataque**: `reproducir_sonido(sound_dive)`
            - Calcular la **dirección hacia el jugador**:
                - `direction = (player.global_position - global_position).normalized()`
FUNCIÓN _dive_state(delta):
```

```
    - SI player EXISTE:
        - **Seguir ajustando dirección hacia el jugador**:
            - `direction = (player.global_position - global_position).normalized()`
        - **Mover en la dirección del jugador**:
            - `velocity = direction * speed`
        - **Voltear sprite en la dirección del jugador**:
            - `sprite.flip_h = direction.x < 0`
    - **Si impacta contra el suelo, explotar**:
        - SI `is_on_floor() == True`:
            - Cambiar `state = EXPLODE`
FUNCIÓN _explode_state():
    - **Evitar explosiones múltiples**:
        - SI `has_exploded == True`: RETORNAR
    - **Marcar explosión**:
        - `has_exploded = True`
        - `velocity = Vector2.ZERO`
        - Reproducir animación `"blast"`
        - **Reproducir sonido de explosión**: `reproducir_sonido(sound_explosion)`
    - **Activar área de daño**:
        - `area2d_damage.monitoring = True`
    - **Esperar a que termine la animación**:
        - `await animation_player.animation_finished`
    - **Almacenar el dragón en el Object Pool en lugar de eliminarlo**:
        - SI node_pool EXISTE:
            - `node_pool.store(self)`
            - MOSTRAR `"Enemigo guardado en el baúl:", self`
        - SI NO:
            - ELIMINAR `queue_free()`
#############################-- DETECCIÓN DE JUGADOR --#############################
FUNCIÓN _detecta_jugador() -> bool:
    - SI `area2d_patrullaje.has_overlapping_bodies()`:
        - PARA cada `body` en `area2d_patrullaje.get_overlapping_bodies()`:
            - SI `body.is_in_group("player_group")`:
                - `player = body`  # Guardar referencia al jugador
                - RETORNAR `True`
    - RETORNAR `False`
#########-- DETECCIÓN DE IMPACTO CON EL JUGADOR --################
FUNCIÓN _on_area_2d_damage_body_entered(body):
    - SI `state == EXPLODE` Y `body.is_in_group("player_group")`:
        - MOSTRAR `"El dragón explotó cerca del nodo:", body.name`
        - **Aplicar retroceso al jugador**:
            - `knockback_force = Vector2(200 * direction.x, -100)`
        - **Aplicar daño al jugador**:
```

```
            - `body.take_damage(damage, knockback_force)`
        - **Desactivar área de daño tras el impacto**:
            - `area2d_damage.monitoring = False`
##############################-- REPRODUCCIÓN DE SONIDO --##############################
FUNCIÓN reproducir_sonido(audio_path):
    - CREAR **nodo de sonido temporal**:
        - `var sound_node = AudioStreamPlayer.new()`
    - **Asignar el sonido** al nodo:
        - `sound_node.stream = audio_path`
    - **Añadir el nodo a la escena**:
        - `add_child(sound_node)`
    - **Reproducir sonido**:
        - `sound_node.play()`
    - **Eliminar el nodo cuando termine el sonido**:
        - `await sound_node.finished`
        - `sound_node.queue_free()`
##############################-- OBJECT POOLING --##############################
FUNCIÓN respawn(new_position: Vector2):
    - SI node_pool EXISTE:
        - OBTENER `enemy = node_pool.retrieve(node_pool.get_node_type(self), new_position)`
        - SI `enemy` NO ES NULL:
            - `enemy.is_active = True`
            - MOSTRAR `"Enemigo reaparecido en:", new_position`
```

Algoritmo IA MEDIA Samuráis 3, 4 y 6

```
INICIALIZAR variables:
    speed = 350.0
    gravity = 500.0
    health = 30
    damage = 10
    direction = 1
    attack_cooldown = 1.5 (Samurai_3) | 0.8 (Samurai_6)
    is_attacking = False
    state = PATROL
    coin_scene = referencia a la escena de monedas
    coin_count = 15
    node_pool = buscar "NodePool" en la escena
    is_active = True
FUNCIÓN _ready():
    SI node_pool ES NULL → MOSTRAR ERROR
    Configurar dirección inicial
    Reproducir animación "run"
LOOP _physics_process(delta):
    SI NOT is_active:
        Reproducir animación "idle"
        RETORNAR
    APLICAR gravedad a velocity.y
    Ejecutar lógica de la **máquina de estados**

    Ejecutar move_and_slide()
MÁQUINA DE ESTADOS:
    SI state == PATROL → LLAMAR _patrol(delta)
    SI state == CHASE → LLAMAR _chase(delta)
    SI state == SEARCH → LLAMAR _search(delta)
    SI state == ATTACK → LLAMAR _attack(delta)
    SI state == DEAD → LLAMAR _dead()
FUNCIÓN _patrol(delta):
    Reiniciar search_timer
    SI detecta jugador → state = CHASE
    MOVER en dirección actual
    Reproducir animación "run"
    SI detecta punto de patrulla:
        Cambiar dirección (direction *= -1)
        Voltear sprite (sprite.flip_h)
        Esperar 0.5s
FUNCIÓN _chase(delta):
    SI player DETECTADO:
```

```
        Calcular dirección hacia el jugador
        SI distancia > 100 → MOVER hacia el jugador
        SI distancia <= 100 → state = ATTACK
    SI NO → state = SEARCH
FUNCIÓN _search(delta):
    velocity.x = 0
    Reproducir animación "idle"
    Incrementar search_timer
    SI search_timer >= search_area_time → state = PATROL
FUNCIÓN _attack(delta):
    Elegir ataque aleatorio:
        - Ataque simple → Reproducir "attack_1"
        - Combo de ataques (Samurai_6) → Reproducir "attack_1", "attack_2", "attack_3" con pausas
        - Aplicar daño al jugador si está en rango

    SI ataque termina → state = CHASE
FUNCIÓN _dead():
    velocity.x = 0
    Reproducir animación "death"
    Desactivar colisiones
    Esperar animación terminada
    SI node_pool EXISTE:
        GUARDAR enemigo en node_pool
        MOSTRAR "Enemigo guardado en el baúl"
    SI NO → ELIMINAR enemigo (queue_free())
FUNCIÓN take_damage(amount, knockback_vector):
    health -= amount
    MOSTRAR "El enemigo ha recibido daño"
    APLICAR knockback a velocity
    Ejecutar move_and_slide()
    Reproducir animación "hurt"
    ESPERAR animación terminada
    SI health <= 0 → LLAMAR die()
    SI NO → Reproducir animación "run"
FUNCIÓN die():
    MOSTRAR "El enemigo ha muerto"
    Reproducir animación "death"
    Desactivar colisiones
    ESPERAR animación terminada
    SI coin_scene EXISTE:
        GENERAR coin_count monedas en la posición del enemigo
        ASIGNAR jugador como objetivo de las monedas
    SI node_pool EXISTE:
```

```
        GUARDAR enemigo en node_pool
        MOSTRAR "Enemigo guardado en el baúl"
FUNCIÓN _on_area_2d_damage_body_entered(body):
    SI body pertenece a "player_group":
        MOSTRAR "El enemigo tocó al nodo"
        APLICAR knockback al jugador con dirección opuesta
        LLAMAR body.take_damage(damage, knockback_force)
FUNCIÓN respawn(new_position):
    SI node_pool EXISTE:
        RECUPERAR enemigo de node_pool y colocarlo en new_position
        ACTIVAR enemigo
        MOSTRAR "Enemigo reaparecido"
```

Algoritmo IA Alta Old Samurái

```
INICIALIZAR variables:
    speed = 350.0
    gravity = 500.0
    jump_force = -600.0
    attack_damage = 20
    dash_speed = 600.0
    heal_amount = 30
    max_heals = 3
    heals_left = max_heals
    health = 500
    direction = 1
    can_attack = True
    state = IDLE
    node_pool = buscar "NodePool" en la escena
    is_active = True
    coin_scene = referencia a la escena de monedas
    coin_count = 35
FUNCIÓN _ready():
    SI node_pool ES NULL → MOSTRAR ERROR
    Configurar dirección inicial
    Reproducir animación "idle"
LOOP _physics_process(delta):
    SI NOT is_active:
        Reproducir animación "idle"
        RETORNAR
    APLICAR gravedad a velocity.y
    Ejecutar lógica de la **máquina de estados**

    Ejecutar move_and_slide()
MÁQUINA DE ESTADOS:
    SI state == IDLE → LLAMAR _idle_state()
    SI state == RUN → LLAMAR _run_state(delta)
    SI state == CHASE → LLAMAR _chase_state(delta)
    SI state == JUMP → LLAMAR _jump_state()
    SI state == ATTACK → LLAMAR _attack_state()
    SI state == DEFEND → LLAMAR _defend_state()
    SI state == DASH → LLAMAR _dash_state()
    SI state == HEAL → LLAMAR _heal_state()
    SI state == DEAD → LLAMAR _dead_state()
FUNCIÓN _idle_state():
    Reproducir animación "idle"
    velocity.x = 0
```

```
    SI detecta jugador → state = CHASE
FUNCIÓN _run_state(delta):
    MOVER en dirección actual
    Reproducir animación "run"
    SI detecta jugador → state = CHASE
FUNCIÓN _chase_state(delta):
    SI player DETECTADO:
        Calcular dirección hacia el jugador
        SI distancia < 100:
            Seleccionar acción: Ataque, Curación o Defensa
        SI detecta obstáculo → state = JUMP
    SI NO → state = RUN
FUNCIÓN _jump_state():
    Reproducir animación "jump"
    Aplicar fuerza de salto
    SI detecta jugador → state = CHASE
    SI NO → state = IDLE
FUNCIÓN _attack_state():
    SI puede atacar:
        Elegir tipo de ataque (golpe simple, combo o especial)
        Aplicar daño y knockback al jugador si está en rango
        ESPERAR animación terminada
    state = CHASE
FUNCIÓN _defend_state():
    Reproducir animación "defend"
    ESPERAR animación terminada
    state = CHASE
FUNCIÓN _dash_state():
    Reproducir animación "dash"
    velocity.x = dirección * dash_speed
    ESPERAR animación terminada
    state = CHASE
FUNCIÓN _heal_state():
    SI heals_left > 0:
        heals_left -= 1
        Reproducir animación "healing"
        health += heal_amount
        ESPERAR animación terminada
    state = CHASE
FUNCIÓN _dead_state():
    velocity.x = 0
    Reproducir animación "death"
    Desactivar colisiones
```

```
    ESPERAR animación terminada
    SI coin_scene EXISTE:
        GENERAR coin_count monedas en la posición del enemigo
        ASIGNAR jugador como objetivo de las monedas
    SI node_pool EXISTE:
        GUARDAR enemigo en node_pool
        MOSTRAR "Enemigo guardado en el baúl"
    SI NO → ELIMINAR enemigo (queue_free())
FUNCIÓN take_damage(amount, knockback_vector):
    SI state == DEFEND:
        MOSTRAR "El enemigo bloqueó el ataque"
        Reproducir animación "defend"
        RETORNAR
    health -= amount
    MOSTRAR "El enemigo ha recibido daño"
    APLICAR knockback a velocity
    Ejecutar move_and_slide()
    Reproducir animación "hurt"
    ESPERAR animación terminada
    SI health <= 0 → LLAMAR die()
    SI NO → Reproducir animación "run"
FUNCIÓN die():
    MOSTRAR "El enemigo ha muerto"
    Reproducir animación "death"
    Desactivar colisiones
    ESPERAR animación terminada
    SI coin_scene EXISTE:
        GENERAR coin_count monedas en la posición del enemigo
        ASIGNAR jugador como objetivo de las monedas
    SI node_pool EXISTE:
        GUARDAR enemigo en node_pool
        MOSTRAR "Enemigo guardado en el baúl"
FUNCIÓN _on_area_2d_attack_body_entered(body):
    SI body pertenece a "player_group" y state == ATTACK:
        Aplicar knockback y daño al jugador
FUNCIÓN respawn(new_position):
    SI node_pool EXISTE:
        RECUPERAR enemigo de node_pool y colocarlo en new_position
        ACTIVAR enemigo
        MOSTRAR "Enemigo reaparecido"
```

ALGORTIMO IA SUPERIOR Elf Ranger

```
###########################-- VARIABLES --##############################
INICIALIZAR variables:
    - **Movimiento y Física:**
        - speed = 200  # Velocidad de movimiento
        - gravity = 1000  # Gravedad aplicada al enemigo
        - jump_force = -700  # Fuerza del salto
    - **Estado del enemigo:**
        - health = 2000  # Salud inicial
        - is_active = False  # Determina si el enemigo está activo
        - can_perform_action = True  # Controla si el enemigo puede realizar acciones
        - is_immune = False  # Indica si está en estado de defensa (bloquea ataques)
    - **Dirección del enemigo:**
        - direction = 1  # Indica hacia dónde está mirando el enemigo
        - is_facing_right = True  # Controla la orientación del sprite
    - **Q-Learning:**
        - q_table = {}  # Diccionario para almacenar la tabla de aprendizaje
        - epsilon = 1.0  # Probabilidad de exploración inicial
        - epsilon_decay = 0.99  # Tasa de disminución de exploración
        - epsilon_min = 0.1  # Límite mínimo de exploración
        - alpha = 0.1  # Tasa de aprendizaje
        - gamma = 0.9  # Factor de descuento
        - reward = 0  # Recompensa en cada iteración
        - previous_state = null  # Estado previo en la Q-Table
        - previous_action = null  # Acción previa en la Q-Table
    - **Object Pooling:**
        - node_pool = obtener "NodePool" de la escena
    - **Generación de monedas:**
        - coin_scene = referencia a la escena de monedas
        - coin_count = 5  # Cantidad de monedas generadas al morir
#####################-- INICIALIZACIÓN (_ready) --#######################
FUNCIÓN _ready():
    - Cargar la **Q-Table** desde un archivo JSON si existe.
    - Conectar las **áreas de detección** para activar al enemigo.
    - Establecer la **dirección inicial** del enemigo.
    - Habilitar **raycasts** para detectar al jugador.
    - SI node_pool ES NULL → MOSTRAR ERROR.
######################-- PROCESAMIENTO (_physics_process) --########################
FUNCIÓN _physics_process(delta):
    SI NOT is_active:
        RETORNAR  # Si el enemigo no está activo, detener toda la lógica.
    # Aplicar **gravedad** para que el enemigo se mantenga en el suelo.
    SI NOT is_on_floor():
```

```
        velocity.y += gravity * delta
    # Obtener el estado actual del entorno en un **vector de características**.
    current_state = get_state()
    # Seleccionar una acción basada en **Q-Learning**.
    action = choose_action(current_state)
    # Ejecutar la acción si está disponible.
    SI can_perform_action:
        perform_action(action)
    # Mover al enemigo en base a su velocidad y colisiones.
    move_and_slide()
    # Guardar el estado previo y actualizar la **Q-Table** con la recompensa obtenida.
    SI previous_state != null and previous_action != null:
        update_q_table(current_state)
    # Reducir la probabilidad de exploración (epsilon) progresivamente.
    epsilon = max(epsilon_min, epsilon * epsilon_decay)
    previous_state = current_state
    previous_action = action
#############################-- MÁQUINA DE ESTADOS --################################
ESTADOS POSIBLES:
    - **MOVE_LEFT:** Moverse a la izquierda y reproducir animación "run".
    - **MOVE_RIGHT:** Moverse a la derecha y reproducir animación "run".
    - **JUMP:** Saltar si está en el suelo.
    - **MELEE_ATTACK:** Realizar un ataque cuerpo a cuerpo.
    - **DEFEND:** Activar estado de defensa (bloqueo de daño).
    - **AIR_ATTACK:** Atacar en el aire si el jugador está arriba.
    - **ARROW_ATTACK:** Disparar flecha si el jugador está en rango.
    - **ARROW_RAIN:** Lluvia de flechas si el jugador está en rango.
    - **ARROW_RAY:** Rayo de flechas si el jugador está en el medio.
#############################-- GESTIÓN DE DAÑO --################################
FUNCIÓN take_damage(amount, knockback_vector):
    SI is_immune:
        MOSTRAR "El enemigo bloqueó el ataque"
        RETORNAR  # Ignorar daño mientras está en defensa.
    - Reducir **health** en la cantidad recibida.
    - Aplicar **knockback** para empujar al enemigo en la dirección opuesta.
    - Reproducir animación "hurt".
    SI health <= 0:
        LLAMAR die()
#############################-- MUERTE Y RESPAWN --################################
FUNCIÓN die():
    - Reproducir animación "dead".
    - Guardar la **Q-Table** en archivo para no perder el progreso de aprendizaje.
    - ESPERAR a que termine la animación.
```

```
- **Generar monedas al morir**:
    - SI coin_scene EXISTE:
        - Crear **coin_count** monedas en la posición del enemigo.
        - Asignar al jugador como objetivo de las monedas.
- **Manejo con Object Pooling**:
    - SI node_pool EXISTE:
        - GUARDAR enemigo en node_pool para reutilizarlo.
        - MOSTRAR "Enemigo guardado en el baúl".
    - SI NO:
        - ELIMINAR enemigo (`queue_free()`).
FUNCIÓN respawn(new_position):
    - SI node_pool EXISTE:
        - RECUPERAR enemigo de node_pool y colocarlo en `new_position`.
        - ACTIVAR enemigo.
        - MOSTRAR "Enemigo reaparecido".
#############################-- Q-LEARNING --################################
FUNCIÓN get_state():
    - Obtener **posición relativa del jugador** respecto al enemigo.
    - Verificar si **raycasts detectan al jugador** a la izquierda o derecha.
    - Revisar si el enemigo está **bloqueado por una pared**.
    - Devolver el **estado en forma de vector** con información del entorno.
FUNCIÓN choose_action(state):
    - SI el estado tiene datos en la **Q-Table**:
        - Elegir la mejor acción con la **máxima recompensa aprendida**.
    - SI NO:
        - Elegir una acción **aleatoria** para exploración.
FUNCIÓN update_q_table(current_state):
    - Obtener **máximo valor de recompensa** del estado futuro.
    - Actualizar la **Q-Table** aplicando la ecuación de Q-Learning:
        Q(s, a) = Q(s, a) + α * (reward + γ * max(Q(s', a')) - Q(s, a))
FUNCIÓN save_q_table():
    - Guardar la **Q-Table** en un archivo JSON para su uso en futuras partidas.
FUNCIÓN load_q_table():
    - Cargar la **Q-Table** desde archivo si existe, o inicializar una nueva si no.
#############################-- SISTEMA DE ACCIONES --#############################
FUNCIÓN perform_action(action):
    MATCH action:
        SI action == MOVE_LEFT:
            - Establecer `velocity.x = -speed` (movimiento hacia la izquierda)
            - Llamar `change_direction(False)` para voltear el sprite si es necesario
            - Reproducir animación `"run"`

        SI action == MOVE_RIGHT:
```

```
            - Establecer `velocity.x = speed` (movimiento hacia la derecha)
            - Llamar `change_direction(True)` para voltear el sprite si es necesario
            - Reproducir animación `"run"`
        SI action == JUMP:
            - SI `is_on_floor() == True`:  # Solo salta si está en el suelo
                - Aplicar `velocity.y = jump_force`
                - Reproducir animación `"jump"`
        SI action == MELEE_ATTACK:
            - Llamar `melee_attack()` (ver más abajo)
        SI action == DEFEND:
            - Llamar `defend()` (ver más abajo)
        SI action == AIR_ATTACK:
            - Llamar `air_attack()` (ver más abajo)
        SI action == ARROW_ATTACK:
            - Llamar `arrow_attack()` (ver más abajo)
        SI action == ARROW_RAIN:
            - Llamar `arrow_rain()` (ver más abajo)
        SI action == ARROW_RAY:
            - Llamar `arrow_ray()` (ver más abajo)
    # **Poner un tiempo de espera entre acciones**
    - Establecer `can_perform_action = False`
    - ESPERAR `get_tree().create_timer(action_cooldown).timeout`
    - Establecer `can_perform_action = True`  # Ahora puede ejecutar una nueva acción
########################-- SISTEMA DE ATAQUE CUERPO A CUERPO --########################
FUNCIÓN melee_attack():
    - Reproducir animación `"melee_attack"`

    - PARA cada `body` en `melee_area.get_overlapping_bodies()`:
        SI `body.name == "Enzo"`:  # Si el jugador está en el rango de ataque
            - **Dar recompensa al enemigo**: `reward += 10000`
            - **Aplicar knockback al jugador**:
                - Crear `knockback_force = Vector2(200 * direction, -100)`
                - Llamar `body.take_damage(50, knockback_force)`
########################-- SISTEMA DE DEFENSA --########################
FUNCIÓN defend():
    - Activar **modo de defensa**: `is_immune = True`
    - Reproducir animación `"defend"`
    - **Verificar si el enemigo bloquea un ataque del jugador**:
        - PARA cada `area` en `defend_area.get_overlapping_areas()`:
            SI `area.name == "AttackArea_uno_punio"` o `"AttackArea_dos_punio"`:
                - **Recompensa por defensa exitosa**: `reward += 20000`
                - MOSTRAR `"Defensa exitosa contra el ataque del jugador"`
    - ESPERAR `get_tree().create_timer(1.0).timeout`
```

```
    - **Desactivar defensa**: `is_immune = False`
    - MOSTRAR `"El enemigo ya no está defendiendo"`
########################-- ATAQUE AÉREO --########################
FUNCIÓN air_attack():
    - Reproducir animación `"air_attack"`

    - PARA cada `body` en `air_area.get_overlapping_bodies()`:
        SI `body.name == "Enzo"`:
            - **Dar recompensa al enemigo**: `reward += 15000`
            - **Aplicar knockback al jugador**:
                - Crear `knockback_force = Vector2(200 * direction, -100)`
                - Llamar `body.take_damage(50, knockback_force)`
########################-- ATAQUE CON FLECHA --########################
FUNCIÓN arrow_attack():
    - Reproducir animación `"arrow_attack"`
    - PARA cada `body` en `mid_area.get_overlapping_bodies()`:
        SI `body.name == "Enzo"`:
            - **Dar recompensa al enemigo**: `reward += 5000`
            - **Aplicar knockback al jugador**:
                - Crear `knockback_force = Vector2(200 * direction, -100)`
                - Llamar `body.take_damage(50, knockback_force)`
    - PARA cada `body` en `flecha_area.get_overlapping_bodies()`:
        SI `body.name == "Enzo"`:
            - **Dar recompensa adicional**: `reward += 5000`
            - **Aplicar knockback al jugador**:
                - Crear `knockback_force = Vector2(200 * direction, -100)`
                - Llamar `body.take_damage(50, knockback_force)`
########################-- ATAQUE LLUVIA DE FLECHAS --########################
FUNCIÓN arrow_rain():
    - Reproducir animación `"arrow_rain"`
    - PARA cada `body` en `rain_area.get_overlapping_bodies()`:
        SI `body.name == "Enzo"`:
            - **Dar recompensa al enemigo**: `reward += 10000`
            - **Aplicar knockback al jugador**:
                - Crear `knockback_force = Vector2(200 * direction, -100)`
                - Llamar `body.take_damage(50, knockback_force)`
########################-- ATAQUE RAYO DE FLECHAS --########################
FUNCIÓN arrow_ray():
    - Reproducir animación `"arrow_ray"`
    - PARA cada `body` en `mid_area.get_overlapping_bodies()`:
        SI `body.name == "Enzo"`:
            - **Dar recompensa al enemigo**: `reward += 25000`
            - **Aplicar knockback al jugador**:
```

```
                - Crear `knockback_force = Vector2(200 * direction, -100)`
                - Llamar `body.take_damage(50, knockback_force)`
########################-- CAMBIAR DIRECCIÓN --########################
FUNCIÓN change_direction(facing_right: bool):
    SI `facing_right != is_facing_right`:
        - **Actualizar dirección del sprite**: `is_facing_right = facing_right`
        - **Voltear el sprite**: `sprite.flip_h = not is_facing_right`

        - **Invertir posición de las áreas de ataque y defensa**:
            - `defend_area.scale.x *= -1`
            - `melee_area.scale.x *= -1`
            - `mid_area.scale.x *= -1`
            - `flecha_area.scale.x *= -1`
            - `back_area.scale.x *= -1`
            - `up_reconocimiento_area.scale.x *= -1`
            - `rain_area.scale.x *= -1`
            - `air_area.scale.x *= -1`
```

ALGORITMOS DE OBJETOS

Algoritmo Baúl

```
##############################-- SISTEMA DE OBJECT POOLING --##############################
### -- VARIABLES GLOBALES -- ###
INICIALIZAR:
    - pools = {}  # Diccionario con pilas de nodos reutilizables
    - storage_positions = {}  # Controla la cantidad de objetos almacenados por tipo
    - STORAGE_START_POS = Vector2(-4000, 2000)  # Posición inicial de almacenamiento
    - X_SPACING = 50  # Espaciado entre objetos de un mismo tipo
    - Y_SPACING = 300  # Espaciado entre diferentes tipos de objetos (nueva fila)
FUNCIÓN _ready():
    - LIMPIAR `storage_positions` al iniciar el nodo
##############################-- OBTENER TIPO DE NODO --##############################
FUNCIÓN get_node_type(node: Node) -> String:
    - SI `node.scene_file_path != ""`:
        - **Extraer el nombre base del archivo de escena** usando `get_file().get_basename()`
    - SI NO:
        - RETORNAR cadena vacía ("")
    - RETORNAR el nombre de la escena original del nodo
##############################-- GUARDAR UN NODO EN EL POOL --##############################
FUNCIÓN store(node: Node):
    - Obtener `type_name` usando `get_node_type(node)`
    - SI `type_name == ""`:
        - MOSTRAR ERROR: `"El nodo no tiene un archivo de escena asignado"`
        - RETORNAR
    - SI `type_name` NO está en `pools`:
        - CREAR una **nueva pila** en `pools[type_name]`
        - ASIGNAR una **posición inicial de almacenamiento**:
            - `storage_positions[type_name] = Vector2(STORAGE_START_POS.x, STORAGE_START_POS.y +
len(pools) * Y_SPACING)`
    - SI `node` YA está en `pools[type_name]`:
        - MOSTRAR MENSAJE: `"El nodo ya está en el baúl. No se guarda nuevamente"`
        - RETORNAR
    - **Desactivar el nodo**:
        - `node.is_active = False`
    - **Colocar el nodo en la posición de almacenamiento**:
        - `node.global_position = storage_positions[type_name]`
    - **Guardar el nodo en la pila correspondiente**:
        - `pools[type_name].append(node)`
    - MOSTRAR MENSAJE:
        - `"Nodo guardado:", node, "| Tipo:", type_name, "| Posición:", node.global_position, "|
Visible:", node.visible`
```

```
- **Actualizar la posición de almacenamiento** para el siguiente nodo del mismo tipo:
    - `storage_positions[type_name].x += X_SPACING`
    - MOSTRAR `"Nueva posición de almacenamiento para", type_name, "→", storage_positions[type_name]`

###############################-- RECUPERAR UN NODO DESDE EL POOL --###############################

FUNCIÓN retrieve(type_name: String, position: Vector2) -> Node:
- SI `type_name` está en `pools` y `pools[type_name]` **NO está vacío**:
    - **Obtener el nodo más reciente** con `pop_back()`
    - **Actualizar su posición** con `node.global_position = position`
    - **Activarlo** con `node.is_active = True`
    - MOSTRAR MENSAJE: `"Nodo reutilizado:", node, "| Tipo:", type_name, "| Nueva posición:", position"`
    - RETORNAR `node`
- SI NO:
    - MOSTRAR MENSAJE: `"No hay nodos disponibles en el baúl para:", type_name`
    - RETORNAR `null`
```

Algoritmo Coins

```
### -- VARIABLES GLOBALES -- ###
INICIALIZAR:
    - speed = 50.0  # Velocidad inicial de la moneda
    - attraction_speed = 300.0  # Velocidad al ser atraída por el jugador
    - life_time = 1.0  # Tiempo antes de empezar la atracción al jugador
    - target_offset = Vector2(0, -70)  # Offset para mover la moneda arriba del jugador
    - target = null  # Referencia al jugador
    - attracted = False  # Indica si la moneda está siendo atraída
    - velocity = Vector2.ZERO  # Velocidad de la moneda
    - sound_effect = cargar("res://assets/sound/coin_pickup.wav")  # Sonido al recoger la moneda
#############################-- INICIALIZACIÓN (_ready) --#############################
FUNCIÓN _ready():
    - **Hacer la moneda invisible al inicio**:
        - `visible = False`
    - **Esperar 0.1 segundos antes de mostrarla**:
        - `await get_tree().create_timer(0.1).timeout`
        - `visible = True`
    - **Configurar dirección inicial aleatoria**:
        - `velocity = Vector2(randf_range(-1, 1), randf_range(-1, 1)).normalized() * speed`
    - **Esperar un tiempo antes de activarse la atracción al jugador**:
        - `await get_tree().create_timer(life_time).timeout`
        - `attracted = True`  # Activar el movimiento hacia el jugador
#############################-- MOVIMIENTO Y ATRACCIÓN (_process) --#############################
FUNCIÓN _process(delta):
    - SI `attracted == True` Y `target != null`:
        - Calcular la **posición de atracción** (`target_position`):
            - `target_position = target.global_position + target_offset`
        - **Actualizar la dirección de la moneda hacia el jugador**:
            - `velocity = (target_position - global_position).normalized() * attraction_speed`
    - **Mover la moneda**:
        - `global_position += velocity * delta`
#############################--      DETECCIÓN      DE      COLISIÓN      (_on_body_entered)      --
#############################
FUNCIÓN _on_body_entered(body):
    - SI `body` **pertenece al grupo `"player_group"`**:
        - **Agregar 1 moneda al jugador**:
            - `body.add_coins(1)`
        - **Reproducir efecto de sonido** (Nuevo):
            - `reproducir_sonido(sound_effect)`
        - **Eliminar la moneda**:
            - `queue_free()`
#############################-- REPRODUCCIÓN DE SONIDO --#############################
```

```
FUNCIÓN reproducir_sonido(audio_path):
    - CREAR **nodo de sonido temporal**:
        - `var sound_node = AudioStreamPlayer.new()`
    - **Asignar el sonido** al nodo:
        - `sound_node.stream = audio_path`
    - **Añadir el nodo a la escena**:
        - `add_child(sound_node)`
    - **Reproducir sonido**:
        - `sound_node.play()`
    - **Eliminar el nodo cuando termine el sonido**:
        - `await sound_node.finished`
        - `sound_node.queue_free()`
```

Algoritmo Shuriken

```
##############################-- SISTEMA DE SHURIKEN --##############################
### -- VARIABLES GLOBALES -- ###
INICIALIZAR:
    - speed = 600  # Velocidad de movimiento del shuriken
    - damage = 15  # Daño que inflige al impacto
    - direction = 1  # Dirección de movimiento (1 = derecha, -1 = izquierda)
    - sound_effect = cargar("res://assets/sound/shuriken_throw.wav")  # Sonido al lanzar el shuriken
##############################-- INICIALIZACIÓN (_ready) --##############################
FUNCIÓN _ready():
    - **Reproducir sonido al instanciar el shuriken**:
        - `reproducir_sonido(sound_effect)`
##############################-- MOVIMIENTO (_process) --##############################
FUNCIÓN _process(delta):
    - **Mover el shuriken en la dirección asignada**:
        - `position.x += direction * speed * delta`
    - **Rotar el shuriken constantemente**:
        - `rotation_degrees += 360 * delta`  # Gira en 360° por segundo
##############################--    DETECCIÓN    DE    IMPACTO    (_on_area_2d_body_entered)    --
##############################
FUNCIÓN _on_area_2d_body_entered(body):
    - **Si impacta contra un enemigo**:
        - SI `body.is_in_group("enemies_group")` Y `body.has_method("take_damage")`:
            - MOSTRAR `"El shuriken tocó al enemigo:", body.name`
            - **Aplicar knockback al enemigo**:
                - `knockback_force = Vector2(200 * direction, -100)`
            - **Aplicar daño**:
                - `body.take_damage(damage, knockback_force)`
    - **Si impacta contra el jugador**:
        - SI `body.is_in_group("player_group")` Y `body.has_method("take_damage")`:
            - MOSTRAR `"El shuriken tocó al jugador:", body.name`
            - **Aplicar knockback al jugador**:
                - `knockback_force = Vector2(200 * direction, -100)`
            - **Aplicar daño**:
                - `body.take_damage(damage, knockback_force)`
    - **Eliminar el shuriken tras el impacto**:
        - `queue_free()`
##############################-- REPRODUCCIÓN DE SONIDO --##############################
FUNCIÓN reproducir_sonido(audio_path):
    - CREAR **nodo de sonido temporal**:
        - `var sound_node = AudioStreamPlayer.new()`
    - **Asignar el sonido** al nodo:
        - `sound_node.stream = audio_path`
```

```
- **Añadir el nodo a la escena**:
    - `add_child(sound_node)`
- **Reproducir sonido**:
    - `sound_node.play()`
- **Eliminar el nodo cuando termine el sonido**:
    - `await sound_node.finished`
    - `sound_node.queue_free()`
```

ANEXO GRÁFICO

Figura 58. Enzo en los primeros ciclos de desarrollo

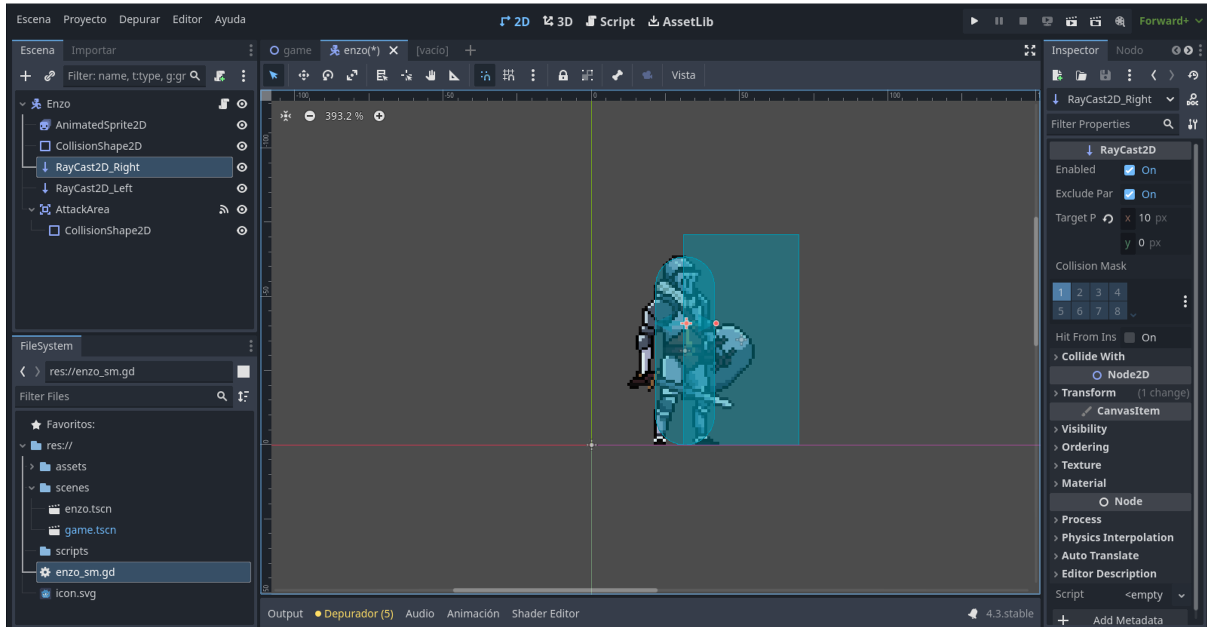


Figura 59. Primer escenario construido

Figura 60. Primer escenario construido (2)

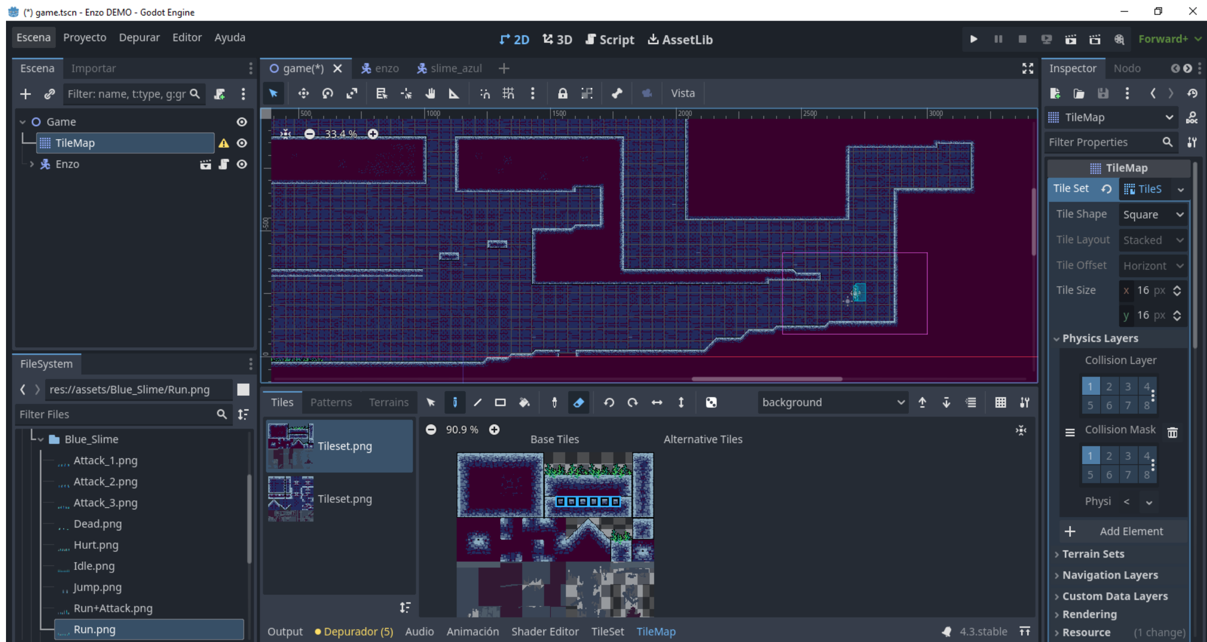


Figura 61. Enemigo de los primeros ciclos del desarrollo

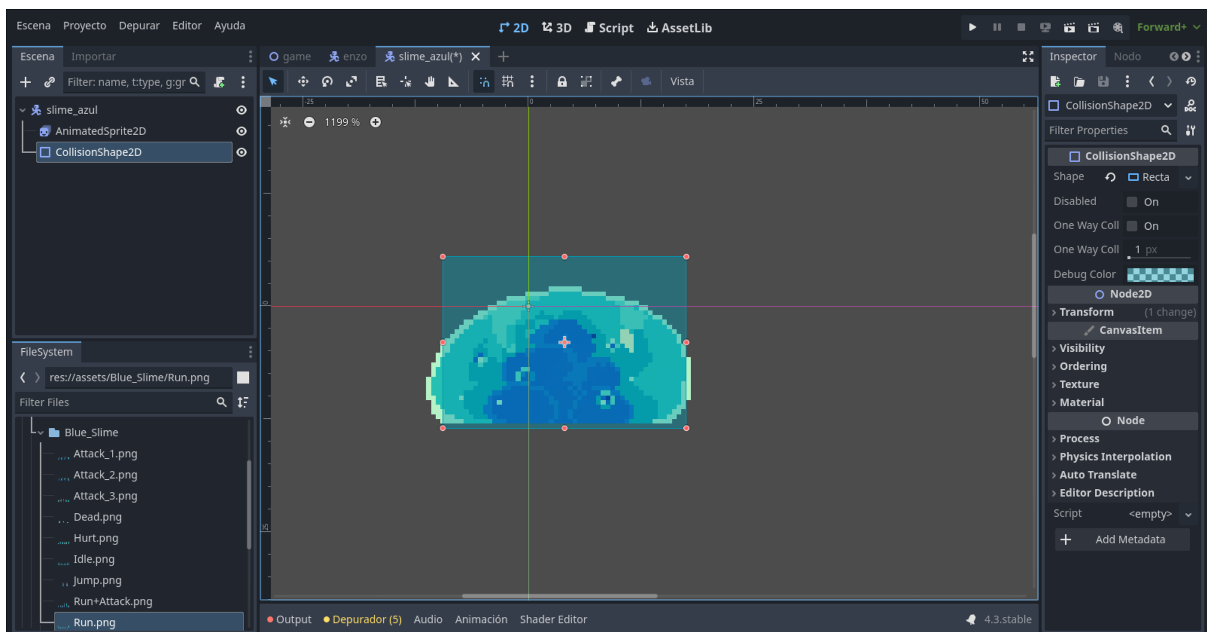


Figura 62. Enemigo de los primeros ciclos del desarrollo (2)

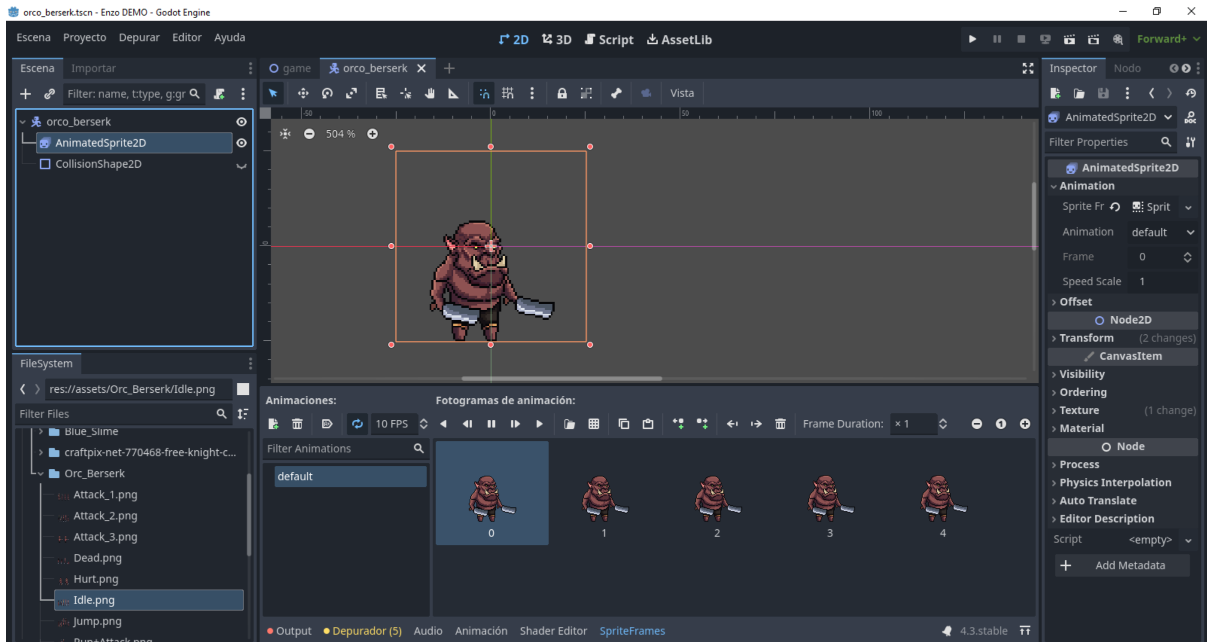


Figura 63. Enemigos considerados en los ciclos iniciales

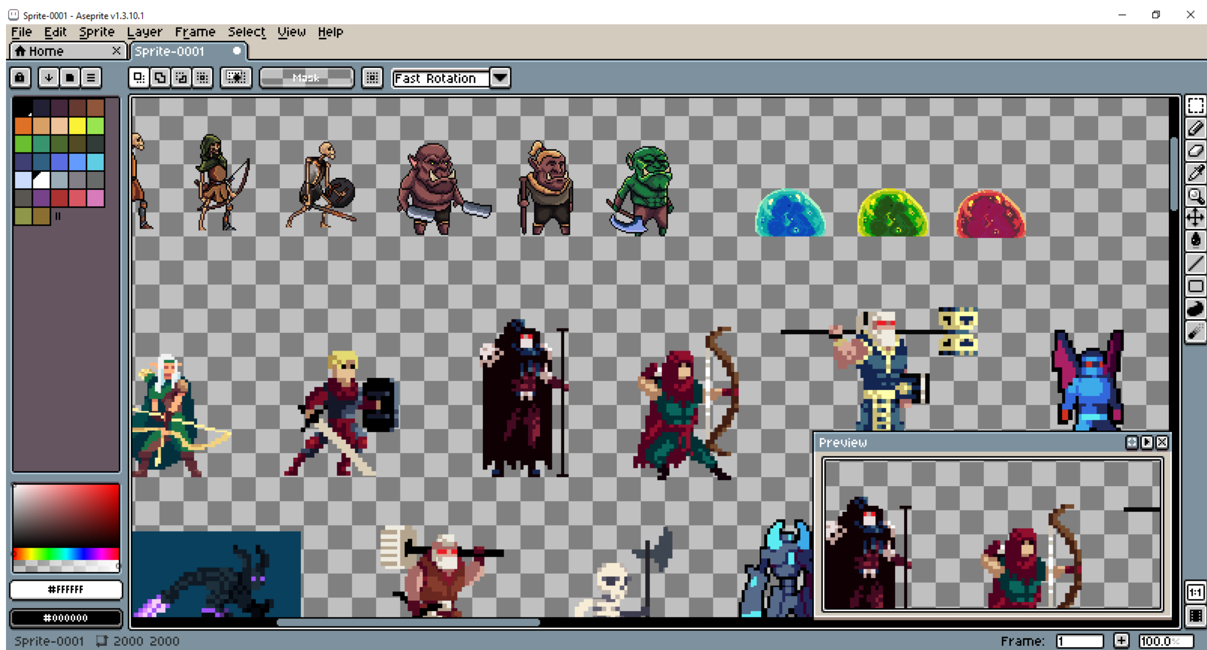
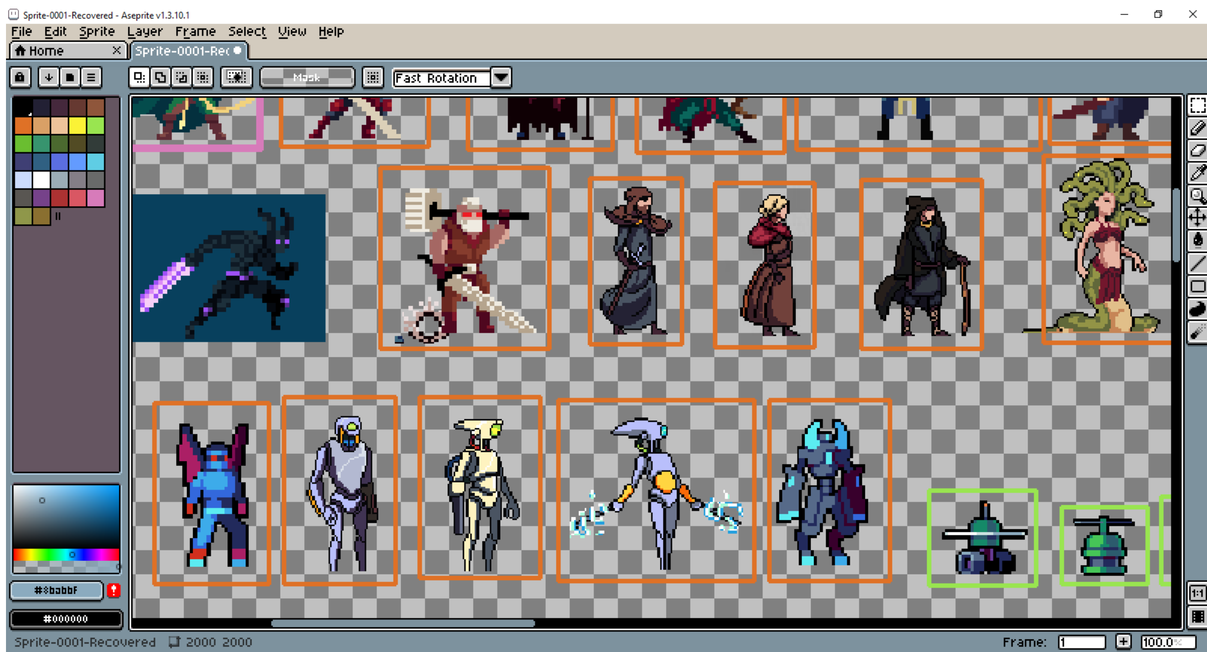


Figura 64. Enemigos considerados en los ciclos iniciales (2)



BIBLIOGRAFÍA

Artículos de noticias y blogs

- Cerdán, F. (2023, 24 de noviembre). *Narrativa del videojuego: Interfaz y experiencia*. Punto de Respawn. https://puntoderespawn.com/articulos/narrativa/narrativa-del-videojuego-interfaz-y-experiencia/
- Redacción EC. (2023, septiembre 14). *Unity empezará a cobrar por el uso de su motor gráfico y los desarrolladores avisan que retirarán juegos de tiendas*. El Comercio. https://elcomercio.pe/tecnologia/videojuegos/unity-empezara-a-cobrar-por-el-uso-de-su-motor-grafico-y-los-desarrolladores-avisan-que-retiraran-juegos-de-tiendas-noticia/?ref=ecr
- Tones, J. (2023, septiembre 18). *Unity comienza a dar marcha atrás: habrá cambios en un par de días en su plan de cobrar por el uso del motor*. Xataka. https://www.xataka.com/videojuegos/unity-comienza-a-dar-marcha-atras-habra-cambios-par-dias-su-plan-cobrar-uso-motor
- Talent Garden. (2023). *Una guía completa sobre la UI en un juego*. Talent Garden. https://blog.talentgarden.com/es/blog/coding/una-guia-completa-ui-juego

Libros y documentos académicos

- Astengo-Noguez, C., & Martínez Elizalde, L. B. (2024). *Una mirada a los algoritmos de IA más comunes: Una clasificación de algoritmos para potenciar la experiencia de un videojuego*. Tecnológico de Monterrey.
- Canosa Ferreiro, A. J. (2024). *SCRUM. Teoría e Implementación práctica*. España: RA-MA S.A. Editorial y Publicaciones.
- Mahdavi, M. (2011). *A Review on Major UI Design Guidelines*. Vusydney. https://www.academia.edu/102637227/A_Review_on_Major_UI_Design_Guidelines
- Rico Zambrana, D. (2016). *Análisis de algoritmos de inteligencia artificial para videojuegos* [Trabajo de grado, Universidad de Málaga]. Universidad de Málaga.
- Universitat Carlemany. (2024, 6 junio). *Estructura de datos: ¿Para qué sirve y qué tipos existen?*. UCMA. https://www.universitatcarlemany.com/actualidad/blog/estructura-datos/

Guías y documentación técnica

- Amazon Web Services. (2024). *¿Qué es el aprendizaje por refuerzo?*. Amazon. https://aws.amazon.com/es/what-is/reinforcement-learning/
- Engine, G. (2025). *Godot Engine - Free and open source 2D and 3D game engine*. Godot Engine. https://godotengine.org/
- Engine, G. (2025). *Community*. Godot Engine. https://godotengine.org/community/
- Engine, G. (2025). *License - Godot Engine*. Godot Engine. https://godotengine.org/license/
- Engine, G. (2025). *Exportando proyectos*. Godot Engine Documentation. https://docs.godotengine.org/es/4.x/tutorials/export/exporting_projects.html
- Scrum Guide. (2025). *Scrum guides*. https://scrumguides.org/scrum-guide.html
- Unity. (2024). *Use object pooling to boost performance with C# scripts in Unity*. Unity Technologies. https://unity.com/es/how-to/use-object-pooling-boost-performance-c-scripts-unity

Recursos y activos para videojuegos

- About Itch.io. (2025). *itch.io*. https://itch.io/docs/general/about
- Godot Asset Library. (2025). https://godotengine.org/asset-library/asset
- Top game assets. (2025). *itch.io*. https://itch.io/game-assets
- Coates, E. (2025). *Game UI Database - Hollow Knight*. Game UI Database. https://www.gameuidatabase.com/gameData.php?id=113

Activos de videojuegos en Itch.io

- 2D Flying Dragon Enemy by Rekaal Games. (2020). *itch.io*. https://rekaalgames.itch.io/2d-flying-enemy
- 200 Free SFX by Kronbits. (2019). *itch.io*. https://kronbits.itch.io/freesfx
- Boss: Demon Slime by chierit. (2021). *itch.io*. https://chierit.itch.io/boss-demon-slime
- Coins 2D by Artist2D3D. (2023). *itch.io*. https://artist2d3d.itch.io/2d
- Free Japanese Fantasy Music Pack by Momiziba. (2024). *itch.io*. https://momiziba.itch.io/free-japanese-music-pack

- Pixel Art GUI Elements by Mounir Tohami. (2020). *itch.io*. https://mounirtohami.itch.io/pixel-art-gui-elements
- Samurai Pack 2D Pixel Art. (2025). *itch.io*. https://itch.io/s/110075/samurai-pack-2d-pixel-art
- Toxic Frog Animations Pixel Art 2D FREE by Eduardo Scarpato. (2023). *itch.io*. https://eduardscarpato.itch.io/toxic-frog-animations-pixel-art-2d-free