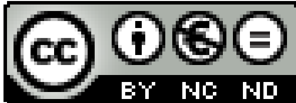
Ascend Tower

Proyecto de desarrollo

CFGM Administración de Sistemas Microinformáticos i Redes

Nathan Ruano y Adrian López
SMX2A
2025-2026



Aquesta obra està subjecta a una llicència de Reconeixement-NoComercial-SenseObraDerivada 3.0 Espanya de Creative Commons

B) GNU Free Documentation License (GNU FDL)

Copyright © ANY Nathan

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU Free Documentation License".

C) Copyright

© (l'autor/a)
Reservats tots els drets. Està prohibit la reproducció total o parcial d'aquesta obra per qualsevol mitjà o procediment, compresos la impressió, la reprografia, el microfilm, el tractament informàtic o qualsevol altre sistema, així com la distribució d'exemplars mitjançant lloguer i préstec, sense l'autorització escrita de l'autor o dels límits que autoritzi la Llei de Propietat Intel·lectual.

Resumen del proyecto:

Ascend Tower es un videojuego de plataformas en 2D que pone a prueba la precisión y la tenacidad del jugador. El proyecto nace de la idea de crear una experiencia arcade donde el protagonista, movido por la ambición de dejar atrás la pobreza, se adentra en una legendaria torre que custodia incalculables tesoros en cada una de sus plantas. El núcleo del juego se basa en un control refinado y una dificultad progresiva a lo largo de **10 niveles**, donde el objetivo principal es recolectar todas las riquezas posibles para alcanzar la fortuna soñada. Con una estética retro inspirada en los efectos VHS y una banda sonora envolvente, el juego ofrece un reto constante de riesgo y recompensa en cada salto.

Para el desarrollo técnico, utilizaremos **Godot** como motor principal de programación y **Suno** para la composición de la musica, profundizando en el dominio de ambas herramientas durante el proceso. El título contará con una campaña principal y un **'Modo Extremo'**, compuesto por **5 niveles adicionales** diseñados para desafiar al jugador con versiones de alta dificultad en salas seleccionadas de la torre.

El desarrollo de Ascend Tower ha consolidado nuestro aprendizaje en Godot y la gestión técnica mediante GitHub. Hemos logrado transformar un reto inicial de conocimiento nulo en un producto pulido, priorizando la precisión mecánica y la fluidez del control. El proyecto demuestra nuestra capacidad para adaptar el alcance a objetivos realistas, entregando un videojuego funcional que equilibra una estética retro con un diseño de niveles desafiante y profesional

Palabras clave:
Plataformas de precisión, Desarrollo de videojuegos 2D, Motor Godot, Estética Retro VHS, Dificultad progresiva, Recolección de tesoros, Modo Extremo

Abstract

Ascend Tower is a 2D platformer video game that tests the player's precision and tenacity. The project was born from the idea of creating an arcade experience where the protagonist, driven by the ambition to leave poverty behind, enters a legendary tower that guards incalculable treasures on each of its floors. The core of the game is based on refined controls and progressive difficulty throughout 10 levels, where the main objective is to collect as much wealth as possible to achieve a dream fortune. With a retro aesthetic inspired by VHS effects and an immersive soundtrack, the game offers a constant challenge of risk and reward in every jump.

For technical development, we used Godot as the primary game engine and Suno for music composition, deepening our mastery of both tools during the process. The title features a main campaign and an 'Extreme Mode,' consisting of 5 additional levels designed to challenge the player with high-difficulty versions of selected rooms in the tower.

The development of Ascend Tower has consolidated our learning in Godot and technical management through GitHub. We have managed to transform an initial challenge of zero knowledge into a polished product, prioritizing mechanical precision and fluid controls. The project demonstrates our ability to adapt the scope to realistic goals, delivering a functional

video game that balances a retro aesthetic with a challenging and professional level design.

Keywords:

Precise Platformer, 2D Game Development , Godot Engine , Retro VHS Aesthetic , Hardcore / Challenging , Treasure Collection , Indie Game, Speedrun-friendly

Nota sobre el uso de Inteligencia Artificial

Durante la elaboración de esta memoria, se han utilizado herramientas de inteligencia artificial generativa como apoyo técnico y lingüístico. El uso de estas herramientas se ha limitado a las siguientes tareas:

- Mejora de la redacción: Optimización de la fluidez del texto y corrección de estilo en el resumen y las conclusiones.

- Traducción técnica: Apoyo en la traducción al inglés de los apartados requeridos para asegurar la precisión terminológica.

- Estructuración de ideas: Ayuda en la organización de conceptos técnicos complejos para facilitar su comprensión lectora.

En todo momento, nosotros, como autores del proyecto, hemos tomado las decisiones técnicas, definido el contenido de cada sección y verificado que lo escrito refleja fielmente el trabajo realizado en *Ascend Tower*. La IA ha actuado como una herramienta de apoyo y soporte, no como un sustituto de nuestro propio criterio, asumiendo nosotros la responsabilidad completa sobre el contenido de este documento.

Gifs en la memoria

La presente memoria técnica incluye diversas ilustraciones en formato GIF para representar visualmente mecánicas clave como el *Coyote Time*, el *Wall Jump* o el comportamiento de las trampas. Sin embargo, al exportar y entregar este documento en formato PDF (formato de lectura estático), las animaciones se detienen y se muestran como imágenes fijas, lo que impide apreciar la fluidez y el "game feel" del proyecto.

Para solventar esta limitación y garantizar una correcta evaluación del trabajo, se ha habilitado un repositorio en la nube que contiene una copia de esta memoria optimizada para su lectura con animaciones.

- Enlace de acceso: [Memoria]

Para una experiencia de lectura completa y fiel al diseño original, se recomienda encarecidamente consultar la memoria a través del enlace proporcionado, donde todos los elementos visuales y dinámicos están plenamente operativos.

Índice

1. Introducción **1**
1.1 Contexto 1
1.2 Justificación 2
1.3 Objetivos 2
1.3.1 Objetivos 2
1.3.2 Objetivos 3
1.4 Estrategia y planificación 3
1.5 Metodología de trabajo 4
1.6 Estudio económico y presupuestario 6
1.6.1 Costos de Personal (Mano de obra) 6
1.6.2 Inversión en Hardware y Software 7
1.6.3 Costos de Mantenimiento y Distribución 7
1.6.4 Resumen del Presupuesto Total 8
1.6.5 Oportunidades de Beneficio y Viabilidad 8
2. Herramientas de desarrollo **9**
2.1 Análisis y requisitos 9
2.1.1 Requisitos funcionales 10
2.1.2 Requisitos no funcionales 11
2.1 Previsión de tareas de investigación 12
2.2 Tecnologías **13**
2.2.2 Comparativa de las tecnologías valoradas 18
2.3 Estructura del projecte 21
2.4 Descripción de los componentes 23
2.4.1 Personaje jugable (CharacterBody2D) 23
2.4.2 Sistema de colisiones y trampas 23
2.4.3 Sistema de menús (Control + VBoxContainer) 24
2.4.4 Sistema de reinicio al morir 24
2.4.5 Shader de efecto retro (CanvasLayer + ColorRect) 24
2.4.6 Sistema de audio (AudioStreamPlayer) 25
2.5 Definición de las tareas 26
2.5.1 Prueba 1 26
2.5.2 Prueba 2 27
2.5.3 Prueba 3 28
2.5.4 Prueba 4 29
2.5.5 Prueba 5 30
2.6 Definición de las funcionalidades **31**
2.6.1 Funcionalidad 1 Movimiento del personaje 31
2.6.2 Funcionalidad 2 Sistema de trampas y reinicio 32
2.6.3 Funcionalidad 3 Modo Extremo 33
2.6.4 Funcionalidad 4 Sistema de menús 34
2.6.5 Funcionalidad 5 Música 35
2.6.6 Funcionalidad 6 Guardar y cargar partida 36

3 Otros capítulos **37**
3.1 Por qué hicimos este proyecto? 37
3.2 Ideas 38
3.3 Juego en general 42
3.4 Godot 43
3.4.1 Primeros diseños de niveles 43
3.4.2 Shader 45
3.4.3 Programación de menús 46
3.4.4 Niveles extremos. 47
3.5 GitHub 48
3.6 Web 49
4 Conclusiones **50**
4.1 Conclusiones generales del proyecto 50
4.2 Consecución de los objetivos 50
4.3 Valoración de la metodología y planificación 51
4.4 Visión a futuro 51
5. Glosario **52**
6. Bibliografia **56**
7 Anexos **58**

Llista de figuras

Figura 1. Logo Godot....13
Figura 2. GDS....13
Figura 3. Logo Github....14
Figura 4. Logo Youtube....14
Figura 5. Logo Itchi io....14
Figura 6. Logo Claude....15
Figura 7. Logo Geminis....15
Figura 8. Logo Drive....15
Figura 9. Logo Suno....16
Figura 10. Logo Gmail....16
Figura 11. Logo OBS....16
Figura 12. Logo Clideo....17
Figura 13. Logo Bongo Cat....17
Figura 14. Movimiento del Personaje....27
Figura 15. Colisiones del Personaje....28
Figura 16. Pasar de Escena y Reiniciar....29
Figura 17. Implementación Shader....30
Figura 18. Efectos de Sonido....31
Figura 19. Funcionalidad 1....32
Figura 20. Funcionalidad 2....33
Figura 21. Funcionalidad 3....34
Figura 22. Funcionalidad 4....35
Figura 23. Funcionalidad 5....36
Figura 24. Funcionalidad 6....37
Figura 25. idea....39
Figura 26. Idea 2....39
Figura 27. Selección personaje....39
Figura 28. Idea juego de terror....40
Figura 29. Idea juego de terror 2....40
Figura 30. Idea juego de terror 3....40
Figura 31. Hollow knight....41
Figura 32. Hollow knight silksong....41
Figura 33. Nine sols....41
Figura 34. Super meat boy....42
Figura 35. Celeste....42
Figura 36. Donkey Kong....42
Figura 37. Castlevania....42
Figura 38. Juego en general....43
Figura 39. Probando las hitbox....44
Figura 40. Ejemplo de mapa inicial....45
Figura 41. Ejemplo de mapa inicial 2....45
Figura 42. Shader de Godot....46

Figura 43. Menú inicio prototipo... 47
Figura 44. Nivel normal..48
Figura 45. Nivel extremo...48
Figura 46. Primer repositorio de pruebas para Github... 49
Figura 47. Repositorio para la web...49
Figura 48. Buzón de sugerencias...50
Figura 49. Pàgina web..50
Figura 50. Script player 1..59
Figura 51. Script player 2..60
Figura 52. Script player 3..61
Figura 53. Script player 4..62
Figura 54. Script sierra.. 63
Figura 55 Contenedor de monedas...65
Figura 56. Envío de opinión...67
Figura 57. Recibir opinión..67
Figura 58. Script De envío de mensajes.. 68
Figura 59. Script de bloques trampa... 69

1. Introducción

El presente proyecto documenta el diseño y desarrollo de un **videojuego de plataformas 2D de corte clásico**, enfocado en la precisión del movimiento y la agilidad del jugador. El objetivo principal ha sido crear una experiencia técnicamente pulida que priorice la jugabilidad y la respuesta inmediata de los controles en un entorno de niveles estructurados.

El concepto final es el resultado de un proceso de **aprendizaje y adaptación técnica**. En las fases iniciales, la propuesta original se orientaba hacia un *tactical shooter 3D* inspirado en títulos como *CS:GO*. Sin embargo, tras una fase de preproducción y análisis de viabilidad, el equipo decidió cambiar el proyecto. Esta decisión se basó en la complejidad del modelado 3D y la animación para un primer contacto con motores de videojuegos, optando por asegurar la finalización y calidad del producto frente a una ambición excesiva.

Posteriormente, el desarrollo evolucionó a través de una fase intermedia inspirada en el género *Metroidvania* (referenciando a *Hollow Knight* y *Celeste*). Durante esta etapa, se detectó que el *moveset* diseñado rápido y extremadamente preciso no lograba una sinergia adecuada con mapas extensos de exploración abierta. Ante este desafío de diseño y la complejidad de implementar patrones de enemigos satisfactorios en entornos no lineales, el proyecto convergió de forma natural hacia su forma actual, un plataformas de niveles definidos donde la mecánica principal es el movimiento del personaje parecido a títulos como *super meat boy*.

Con el objetivo de ofrecer una experiencia completa, el juego incluirá un modo de dificultad extrema para los jugadores más experimentados.

Somos conscientes del reto que esto representa, especialmente al ser nuestro primer contacto con el motor **Godot**. Partir desde un conocimiento inicial nulo implica un compromiso de aprendizaje acelerado y un esfuerzo extra para transformar esta falta de experiencia previa en un producto entretenido y con el que nos quedemos conformes.

1.1 Contexto

El proyecto se inspira en el género de los **plataformas de precisión**, una tendencia actual en el desarrollo independiente popularizada por títulos como Super Meat Boy o Celeste.

Inicialmente, el desarrollo exploró mecánicas de tactical shooter y metroidvania (inspirado en Hollow Knight). No obstante, durante la preproducción se observó que un moveset rápido y preciso no alcanzaba una sinergia adecuada con mapas de exploración abierta. Por lo tanto, el proyecto ha evolucionado hacia un formato de niveles definidos donde la mecánica principal es la destreza del personaje, siguiendo los estándares de fluidez y reto que definen al género en la actualidad.

1.2 Justificación

Hemos decidido sacar adelante este proyecto por tres razones principales:

Valor añadido y rejugabilidad

- La implementación de un **modo de dificultad extrema** busca satisfacer a los jugadores más exigentes, aportando un incentivo adicional para dominar las mecánicas del juego

Asegurar un buen resultado

- Al cambiar el diseño de 3D a 2D, podemos centrar todo nuestro esfuerzo en que el movimiento y los niveles funcionen a la perfección. Es preferible entregar un juego 2D bien pulido y divertido que uno 3D inacabado o con errores.

Aprendizaje real

- Este proyecto es nuestra oportunidad para aprender a usar herramientas profesionales como **Godot** desde cero. Aunque no teníamos experiencia previa, el esfuerzo que estamos haciendo para dominar el motor nos permite demostrar que podemos aprender rápido y crear un producto del que nos sintamos orgullosos.

1.3 Objetivos

El propósito de este proyecto es diseñar y desarrollar un videojuego de plataformas 2D funcional, que destaque por un control preciso y un reto atractivo para el usuario. Esto se divide en las siguientes metas:

1.3.1 Objetivos

Desarrollar un videojuego de plataformas 2D completo

- Crear una experiencia de juego sólida que combine mecánicas de salto, diseño de niveles y una dificultad bien ajustada.

Capacitación en el motor de juegos

- Aprender a utilizar desde cero las herramientas de **Godot** para convertir una idea teórica en un programa funcional.

Implementar una plataforma de distribución

- Crear una página web propia alojada en un servidor para que los usuarios puedan conocer el proyecto y descargar el juego fácilmente.

1.3.2 Objetivos

Aprender Godot

- Aprender a programar las mecánicas principales y el comportamiento del juego dentro del motor.

Lograr un control fluido

- Diseñar un sistema de movimiento que responda de forma inmediata para que el manejo del personaje sea satisfactorio.

Crear niveles con dificultad progresiva

- Diseñar escenarios que enseñen al jugador las mecánicas básicas y aumenten el reto a medida que avanza.

Desarrollar un "Modo Extremo"

- Añadir una variante de alta dificultad para ofrecer un desafío extra a los jugadores con más experiencia.

Publicar el proyecto en la web

- Configurar el hosting y los enlaces necesarios para que el juego esté disponible para su descarga en internet.

1.4 Estrategia y planificación

Para llevar a cabo este trabajo, se han valorado diferentes rutas, desde la modificación de plantillas existentes hasta el desarrollo íntegro. A continuación, se detalla la elección final y su justificación técnica.

Estrategia elegida

- La estrategia seleccionada es el **desarrollo de un producto nuevo**. En lugar de adaptar un juego ya hecho, hemos decidido crear el videojuego desde cero utilizando el motor **Godot**. Esta decisión nos permite tener un control total sobre las mecánicas de movimiento y el diseño de los niveles, garantizando que el juego se ajuste exactamente a la experiencia de "plataformas de precisión" que queremos ofrecer.

Estudio de viabilidad

Consideramos que esta es la estrategia más apropiada por los siguientes motivos:

Adaptación técnica

- Inicialmente planteamos un proyecto en 3D, pero tras analizar los recursos y el tiempo disponible, determinamos que un entorno 2D es mucho más viable. Esto nos permite asegurar un producto final de mayor calidad y sin errores técnicos graves.

Curva de aprendizaje

- Al desarrollar desde cero, el equipo se obliga a entender cada parte del código y del diseño en Godot. Esto garantiza que cualquier problema que surja durante el desarrollo pueda ser resuelto por nosotros mismos, ya que conocemos el proyecto desde su base.

Gestión de recursos

- Al no depender de modelos o sistemas complejos de terceros, podemos centrarnos en lo que realmente importa para nuestro objetivo, que el personaje se mueva bien y los niveles sean divertidos.

En resumen, el paso del 3D al 2D y la creación propia desde cero es la opción más segura para cumplir con los plazos de entrega y lograr un resultado lo más profesional posible.

1.5 Metodología de trabajo

Para el desarrollo de este videojuego, seguiremos una **metodología ágil basada en el marco de trabajo Scrum**, adaptada a las dimensiones de nuestro equipo. A diferencia de los modelos tradicionales (como el modelo en cascada o "waterfall"), la metodología ágil nos permite trabajar de forma flexible y realizar ajustes sobre la marcha.

Justificación de la metodología

Consideramos que este enfoque es el más apropiado por los siguientes motivos:

Capacidad de adaptación

- Como hemos experimentado en las fases iniciales, el diseño de un juego puede cambiar al probar las mecánicas (el paso del 3D al 2D). Una metodología ágil nos permite pivotar y mejorar el proyecto sin tener que reiniciar toda la planificación.

Entregas incrementales

- Dividiremos el trabajo en pequeños ciclos. Esto nos permite tener una versión jugable lo antes posible (un prototipo) e ir añadiendo niveles y el modo extremo de forma progresiva.

Gestión del aprendizaje

- Al ser nuestro primer contacto con **Godot**, necesitamos una metodología que nos permita aprender y aplicar esos conocimientos inmediatamente, evaluando los resultados cada semana.

Herramientas de seguimiento

- Para asegurar que el proyecto avance según lo previsto y cumplir con los plazos, utilizaremos las siguientes herramientas:

Trello (Tablero Kanban)

Utilizaremos esta aplicación para organizar las tareas en columnas: "Por hacer", "En proceso" y "Finalizado". Esto nos da una visión clara de en qué está trabajando cada miembro del equipo en todo momento.

Diagrama de Gantt

- Aunque usemos agilidad, emplearemos un diagrama de Gantt simplificado para marcar las fechas límite de las fases principales (diseño, programación de mecánicas, creación de niveles y lanzamiento web).

Reuniones de control

- Realizaremos puestas en común periódicas para revisar el progreso del juego, probar las nuevas funcionalidades y decidir los siguientes pasos a seguir.

1.6 Estudio económico y presupuestario

El objetivo de este estudio es determinar el coste teórico de desarrollo, mantenimiento y las posibles vías de monetización del videojuego. Aunque este proyecto se realiza en un entorno académico, se desglosan los gastos como si fuera un encargo profesional para evaluar su viabilidad comercial.

1.6.1 Costos de Personal (Mano de obra)

Se calcula el coste basado en un equipo de dos desarrolladores trabajando durante un ciclo de desarrollo estimado (aprox. 300 horas totales entre ambos).

Perfil	**Horas**	**Precio/Hora (estimado)**	**Total**
Desarrollador de videojuegos (Godot/GDScript)	200h	25 €/h	5.000 €
Diseñador de Niveles y QA	100h	20 €/h	2.000 €
Total Personal			**7.000 €**

1.6.2 Inversión en Hardware y Software

El proyecto se ha beneficiado del uso de herramientas de código abierto para minimizar gastos iniciales.

Software de desarrollo (Godot Engine)

- 0 € (Licencia MIT gratuita).

Composición Musical (Suno AI)

- 10 €/mes (Suscripción Pro para derechos comerciales).

Hardware (PCs existentes)

- Amortización estimada de equipos por el periodo del proyecto: 200 €.

Total Activos: 210 €

1.6.3 Costos de Mantenimiento y Distribución

Para cumplir con el objetivo de publicar el juego en la web, se deben considerar los siguientes costes anuales:

Hosting y Dominio Web

- 60 €/año.

Plataformas de distribución (Ej. Itch.io o Steam)

- Steam requiere un pago único de aprox. 90 € por juego (Itch.io es gratuito)

Total Mantenimiento (Año 1): 150 €

1.6.4 Resumen del Presupuesto Total

Concepto	**Coste**
Mano de obra	7.000 €
Material y Software	210 €
Mantenimiento y Lanzamiento	150 €
TOTAL DEL PROYECTO	**7.360 €**

1.6.5 Oportunidades de Beneficio y Viabilidad

Dada la naturaleza del proyecto el modelo de negocio sugerido es **Premium** o **Donaciones** (Freemium en Itch.io).

Conclusión de viabilidad

- El proyecto es altamente viable debido al uso de herramientas gratuitas como Godot y Suno, lo que permite que casi el 100% de la inversión sea tiempo de desarrollo.

Es importante precisar que las cifras presentadas en este estudio económico tienen un carácter orientativo y se basan en una estimación general del mercado laboral actual. Debido a la complejidad que supone determinar salarios netos exactos y costes de seguridad social detallados sin una estructura empresarial real, los valores reflejados deben entenderse como una aproximación teórica. El objetivo de este apartado es ofrecer una visión global del coste humano y técnico que supondría el desarrollo del proyecto en un entorno profesional, sin pretender ser un balance contable estrictamente exacto

2. Herramientas de desarrollo

2.1 Análisis y requisitos

Nos propusimos crear un juego de plataformas 2D con una jugabilidad fluida, cómoda y precisa, inspirado en los clásicos de Konami y en la exigencia mecánica de títulos como Celeste y Super Meat Boy. Para considerar el desarrollo del proyecto como finalizado, el producto debe cumplir con los siguientes requisitos fundamentales que garantizan su funcionalidad técnica.

Flujo de juego completo

- El jugador debe poder iniciar una partida, superar los retos y llegar al final sin errores críticos.

Contenido estructural

- Implementación total de la torre con sus plantas funcionales, asegurando que el "Modo Extremo" sea totalmente accesible.

Integración de activos

- Todos los elementos visuales (sprites) y sonoros (banda sonora de Suno) deben estar correctamente vinculados en el motor Godot.

Publicación y acceso

- El proyecto no se considera finalizado hasta que esté disponible para descarga mediante una página web propia y su servidor configurado.

2.1.1 Requisitos funcionales

Los requisitos funcionales describen los servicios, tareas o funciones específicas que el sistema debe ejecutar. Representan el "qué" hace el videojuego y cómo reacciona a las entradas del usuario.

Mecánicas de movimiento

- El sistema debe permitir el desplazamiento lateral, el salto preciso y la respuesta inmediata de los controles.

Sistema de colisiones e interacción

- Detección exacta de impactos con pinchos, trampas y obstáculos que dificulten el progreso o causen la muerte del personaje.

Gestión de estados de partida

- Inclusión de un sistema que permita reiniciar el nivel automáticamente al morir

Interfaz de usuario (UI)

- Disponibilidad de menús operativos para el inicio y la pausa durante la partida

2.1.2 Requisitos no funcionales

Los requisitos no funcionales definen las propiedades, restricciones o atributos de calidad que el sistema debe poseer. No se refieren a funciones específicas, sino al "cómo" funciona el sistema.

Rendimiento y fluidez

- La ejecución debe mantener una tasa mínima de 60 FPS para no perjudicar la precisión del control.

Latencia de control

- Los controles deben ser responsivos y sin retraso perceptible para garantizar la agilidad requerida en el género.

Optimización de carga

- Las transiciones entre escenas o niveles deben realizarse en un tiempo inferior a los 3 segundos.

Compatibilidad de sistema

- El software debe ejecutarse de forma nativa tanto en sistemas operativos Windows como en Linux.

Usabilidad

- La interfaz de usuario debe ser clara, intuitiva y fácil de utilizar para cualquier perfil de jugador.

Identidad estética

- Mantenimiento de una coherencia visual de estilo retro (pixel art) inspirada en los juegos clásicos de Konami.

Estabilidad sonora

- El audio y la música deben reproducirse de forma continua, sin fallos técnicos ni interrupciones bruscas.

2.1 Previsión de tareas de investigación

En este apartado se detallan aquellas tareas de análisis y estudio técnico que deben realizarse para asegurar la viabilidad del proyecto. El resultado de estas investigaciones permitirá validar si las tecnologías elegidas son capaces de soportar las mecánicas de alta precisión planificadas.

Aprender a controlar el movimiento en Godot

- Investigar cómo programar al personaje para que se mueva de forma suave, salte bien y no tenga retrasos. El objetivo es que se sienta tan bien como en los juegos profesionales (*Celeste* o *Super Meat Boy*).

Investigar el sistema de saltos y rebotes

- Probar diferentes códigos en **GDScript** para ver cuál funciona mejor para hacer saltos largos, cortos o rebotar en las paredes de forma ágil.

Aprender a organizar el juego por escenas

- Entender cómo separar cada nivel del juego en escenas distintas dentro de Godot. Así, si cometemos un error en un nivel, no romperemos el resto del juego.

Configurar las trampas y la muerte

- Probar cómo hacer que los pinchos y los obstáculos detecten al jugador al instante, para que si lo tocan, el nivel se reinicie solo y sin fallos.

Optimizar el juego

- Investigar cómo hacer que los gráficos (pixel art) se vean bien pero que el juego vaya siempre muy rápido (a 60 FPS) y no se trabe en ningún ordenador.

Integrar la música de la IA

- Aprender a meter las canciones creadas con Suno en el motor de juego para que la música suene siempre de fondo y cambie correctamente al pasar de nivel.

Investigar cómo subir el juego a internet

- Buscar la mejor forma de configurar nuestro servidor web para que cualquier persona pueda entrar en nuestra página y descargarse el juego fácilmente.

2.2 Tecnologías

Godot engine

Figura 1. Logo Godot

Para el desarrollo del núcleo del videojuego, hemos seleccionado **Godot Engine**, un motor de código abierto y multiplataforma altamente flexible. Se ha elegido específicamente por su excelente rendimiento en el ámbito de los juegos en **2D**, permitiendo crear una experiencia visual fluida que puede ser desplegada en sistemas operativos como **Windows y Linux**, cumpliendo así con nuestros requisitos de portabilidad.

Las funcionalidades de Godot nos han permitido crear un juego dinámico mediante su sistema de nodos. Aunque el motor soporta múltiples lenguajes, su arquitectura está optimizada para el desarrollo de plataformas de precisión, facilitando la creación de niveles complejos con una respuesta inmediata a los controles del jugador.

En lo que se refiere al **tratamiento de colisiones**, Godot ha sido fundamental para nuestro proyecto. Permite la creación de estructuras de colisión precisas utilizando formas primitivas (rectángulos y círculos), lo cual es vital para detectar correctamente el contacto con pinchos y trampas, evitando errores de detección que puedan frustrar al usuario.

Además de la lógica de juego, hemos aprovechado las capacidades de Godot para la **reproducción de audio** y el diseño de **interfaces gráficas (UI)**, integrando de forma sencilla las pistas musicales generadas con **Suno AI** y los menús de inicio, pausa y derrota.

GDScript

Para la programación de las mecánicas y la lógica de los 20 niveles, hemos escogido **GDScript**, el lenguaje nativo integrado en Godot. Se trata de un lenguaje de alto nivel, imperativo y orientado a objetos, con una sintaxis clara similar a Python, lo que facilita enormemente su aprendizaje y uso.

Figura 2. Logo GDS

Github

Para la gestión del proyecto y el trabajo colaborativo, hemos utilizado **GitHub**, una plataforma de alojamiento de código basada en Git. Su uso ha sido fundamental para permitir el desarrollo conjunto entre los miembros del equipo:

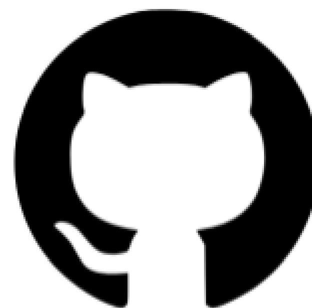


Figura 3. Logo Github

Control de versiones: Nos ha permitido subir los archivos del juego de forma organizada, creando un historial de cambios.

Trabajo Sincronizado: Ha facilitado que, cuando un miembro del equipo terminaba una tarea y subía los avances, el otro pudiera descargar la versión más reciente para continuar trabajando sin riesgo de perder información o duplicar archivos.

Youtube

Esta plataforma ha sido nuestra principal **fuente de documentación visual y aprendizaje**. Debido a la gran comunidad que respalda a Godot Engine, hemos utilizado YouTube para:

Figura 4. Logo Youtube

Formación técnica: Visionado de tutoriales para aprender el manejo del motor desde cero y dominar el lenguaje GDScript.

Inspiración y diseño: Análisis de otros juegos de plataformas para extraer ideas sobre mecánicas de salto, diseño de niveles y estéticas visuales que encajaran con nuestro proyecto.

Itch.io

Como plataforma de distribución, hemos elegido **Itch.io**, uno de los portales más importantes para desarrolladores independientes.

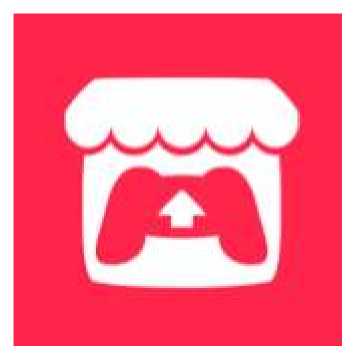


Figura 5. Logo Itch io

Alojamiento web: La hemos utilizado para subir los archivos ejecutables de nuestro juego y configurar la página de descarga.

Accesibilidad: Gracias a esta herramienta, el videojuego es accesible para el público, permitiendo que cualquier usuario pueda jugarlo directamente o descargarlo, cumpliendo así con nuestro objetivo de publicación

Claude

En cuanto al apoyo en la toma de decisiones y resolución de problemas, hemos integrado el uso de **Claude**, una inteligencia artificial avanzada.

Figura 6. Logo Claude

Consultoría técnica: La hemos empleado como una herramienta de búsqueda rápida de información y para resolver dudas puntuales sobre la lógica del juego.

Corrección y redacción: Ha sido clave para revisar y corregir errores en los textos del proyecto, así como para asegurar que la documentación y la memoria técnica mantengan una estructura profesional y coherente.

Geminis

Al igual que Claude, hemos integrado **Gemini** (la inteligencia artificial de Google) como un pilar de apoyo en el desarrollo.

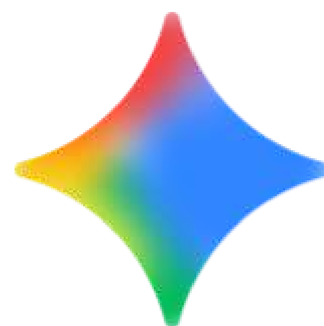


Figura 7. Logo Geminis

Disponibilidad y apoyo continuo: Se ha utilizado principalmente debido a su ausencia de límites de uso gratuito, lo que nos ha permitido realizar consultas extensas y constantes.

Resolución de dudas: Ha servido para obtener explicaciones técnicas adicionales y como una segunda opinión para contrastar soluciones de código en GDScript.

Google Drive

Para la gestión documental y la organización del proyecto, hemos utilizado **Google Drive**.

Figura 8. Logo Drive

Redacción colaborativa: Ha sido la herramienta principal para la creación y edición de este documento de memoria, permitiendo que ambos miembros del equipo pudiéramos escribir y corregir el contenido de forma simultánea y en tiempo real.

Almacenamiento en la nube: Nos ha servido para centralizar toda la documentación teórica y esquemas de diseño en un solo lugar seguro y accesible desde cualquier dispositivo.

Suno AI

El apartado sonoro ha sido cubierto mediante **Suno AI**, una plataforma avanzada de generación de audio mediante inteligencia artificial.

Figura 9. Logo Suno

Composición musical: Se ha utilizado para crear la banda sonora original del juego, buscando ritmos que encajaran con la tensión de un juego de plataformas de precisión.

Ambientación: Gracias a esta herramienta, hemos podido dotar a cada planta de la torre de una atmósfera única, reforzando la identidad retro del proyecto sin necesidad de recursos externos complejos.

Gmail

Para la comunicación directa y el intercambio de recursos rápidos, hemos hecho uso de **Gmail**.

Figura 10. Logo Gmail

Intercambio de activos: Se ha utilizado como canal secundario para compartir imágenes, bocetos y recursos gráficos de forma rápida antes de integrarlos en la estructura final del proyecto.

Comunicación formal: Ha servido para mantener un registro de los envíos de materiales importantes entre los miembros del equipo.

OBS studio

Se ha empleado como software principal para la captura de vídeo de alta fidelidad.

Figura 11. Logo OBS

Su función ha sido grabar sesiones de *gameplay* en tiempo real, permitiendo obtener el material base necesario para mostrar las mecánicas de juego y el diseño de niveles de forma visual en este documento.

Clideo

Actúa como la herramienta de postproducción y edición de vídeo.

Se ha utilizado para procesar las capturas realizadas, realizar recortes precisos de las secuencias y, fundamentalmente, para la exportación y conversión del contenido de vídeo a formato GIF, permitiendo así la integración de imágenes con movimiento dentro de la memoria para una mejor comprensión de las dinámicas de juego.

Figura 12. Logo Clideo

Bongo Cat MR

Esta aplicación se ha utilizado de forma complementaria durante las grabaciones de OBS para superponer una interfaz visual que muestra las pulsaciones de teclado y movimientos del ratón en tiempo real. Su uso es clave para demostrar la precisión de los controles y la respuesta técnica del personaje ante los comandos del jugador.

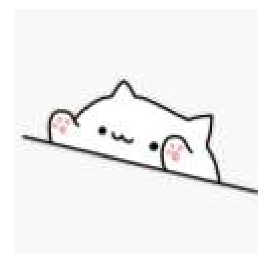


Figura 13. Logo Bongo Cat

2.2.2 Comparativa de las tecnologías valoradas

Durante la fase de planificación e investigación, se analizaron diferentes alternativas tecnológicas antes de decidir las herramientas finales. El objetivo fue encontrar el equilibrio entre potencia técnica, facilidad de uso y coste cero para el proyecto.

Motores de juego: Godot vs. Unity vs. Unreal Engine

Godot Engine (Elegido)

- Se seleccionó por ser un motor de código abierto (totalmente gratuito), extremadamente ligero y con un sistema de nodos ideal para juegos en 2D. Su lenguaje, GDScript, permite un desarrollo mucho más rápido y fluido para un equipo pequeño que otros motores más pesados.

Unity

- Se descartó debido a su mayor complejidad y a los recientes cambios en sus políticas de licencias. Además, requiere un hardware más potente para trabajar cómodamente, algo que no era necesario para nuestro plataformas 2D.

Unreal Engine

- Aunque es un motor líder, está muy enfocado al desarrollo 3D de alta gama. Para un proyecto de plataformas de precisión, su curva de aprendizaje y peso resultaban excesivos.

Asistentes de IA: Claude y Gemini vs. Chat GPT

Claude y Gemini (Elegidos)

- Se prefirieron por su capacidad de razonamiento y programación. **Gemini** fue clave gracias a su ausencia de límites en el uso gratuito, lo que permitió trabajar sin interrupciones. **Claude** se mantuvo como apoyo por su precisión en la redacción y corrección de la documentación técnica.

Chat GPT (Modelos gratuitos)

- Aunque es una herramienta útil, durante las pruebas de desarrollo, los modelos gratuitos de Claude y Gemini ofrecieron respuestas más actualizadas y precisas sobre la versión 4.6 de Godot.

Audio: Suno AI vs. Librerías de recursos

Suno AI (Elegido)

- Nos ha permitido crear una banda sonora 100% original que se adapta al ritmo de cada nivel de la torre. Esto evita el uso de canciones genéricas que ya han sido utilizadas en muchos otros proyectos.

Librerías externas

- Se descartaron para evitar perder tiempo buscando música que encajara con la estética del juego y para garantizar que toda la ambientación sonora fuera exclusiva de nuestro proyecto.

Gestión y Comunicación: GitHub vs. Métodos manuales

GitHub (Elegido)

- Es la herramienta profesional estándar para el trabajo en equipo. Permite subir y bajar cambios de forma segura, evitando que un miembro del equipo borre o sobrepase el trabajo del otro.

Google Drive/Gmail (Uso secundario)

- Se descartaron como herramientas principales para el código, ya que intercambiar archivos a mano genera muchos errores de sincronización en los proyectos de Godot. Se mantuvieron solo para documentos de texto e imágenes rápidas.

Publicación del proyecto steam vs itch.io

Itch.io (Elegida)

- Es la plataforma seleccionada para distribuir el juego debido a su gratuidad y facilidad de uso. A diferencia de otras tiendas, no requiere cuotas de alta, adaptándose perfectamente a un proyecto académico. Además, permite la ejecución directa desde el navegador, lo que facilita que el tribunal o cualquier usuario pruebe el juego al instante sin necesidad de descargas.

Steam

- Se valoró como la alternativa comercial estándar por su gran volumen de usuarios y herramientas de marketing profesional. Sin embargo, se descartó porque exige el pago de una tasa de publicación (Steam Direct) y un proceso de revisión de varios días. Priorizamos la agilidad y el presupuesto cero de **Itch.io** para cumplir con los plazos del desarrollo.

2.3 Estructura del projecte

La arquitectura de *Ink Tower* se basa en el sistema de **Escenas y Nodos** propio de Godot Engine, lo que permite una estructura modular y escalable. El proyecto se organiza en tres capas principales que interactúan entre sí para ofrecer la experiencia de juego:

Esquema Conceptual de la Torre

El juego se estructura como un conjunto de niveles independientes que el jugador debe superar de forma secuencial.

Escena Principal

- Actúa como el nodo raíz que gestiona el cambio entre los diferentes niveles y los menús globales (inicio, pausa, derrota).

Niveles

- Cada planta de la torre es una escena independiente (del nivel 1 al 10). Esto permite que un error en un nivel no afecte la estabilidad de los demás.

Modo Extremo

- Una variante lógica que reutiliza las escenas de los niveles pero con parámetros de dificultad y obstáculos modificados.

Diagrama de Comunicación de Componentes

Para que el juego funcione, los distintos elementos se comunican siguiendo este flujo:

Entrada del Usuario (Input)

- El jugador pulsa las teclas de movimiento o salto.

Lógica del Personaje

- Un script en GDScript procesa estas entradas y calcula las físicas de movimiento, gravedad y colisiones en tiempo real.

Gestor de Colisiones (Physics Engine)

- Godot detecta si el jugador toca un elemento del entorno (suelo) o una trampa (pinchos/sierras).

Sistema de Estados

- Si hay contacto con una trampa, el sistema envía una señal para **reiniciar el nivel**. Si el jugador toca la moneda , se carga la **siguiente escena** de la torre.

Tecnologías en la Estructura

- **GDScript:** Se utiliza para unir todas las piezas, controlando desde el movimiento del personaje hasta la lógica del cronómetro.

- **Nodos de Colisión 2D:** Utilizados para definir las zonas seguras y peligrosas de cada planta de forma precisa.

- **Sistema de Señales (Signals):** Se emplea para que los objetos (como las trampas o la meta) avisen al gestor del juego cuando el jugador interactúa con ellos, manteniendo un código limpio y organizado.

2.4 Descripción de los componentes

A continuación se describen todos los componentes principales que conforman la arquitectura de Ascend Tower, especificando su función dentro del proyecto y las tecnologías de Godot utilizadas en cada caso.

2.4.1 Personaje jugable (CharacterBody2D)

El personaje es el componente central del juego. Está implementado sobre un nodo **CharacterBody2D**, que es el tipo de nodo de Godot diseñado específicamente para personajes controlados por el jugador que necesitan interactuar con la física del mundo.

Su comportamiento está controlado por un script en **GDScript** que gestiona en tiempo real la entrada del teclado, el movimiento horizontal, la gravedad, el salto y las animaciones correspondientes a cada estado (idle, correr, saltar, caer). Para que el movimiento se sienta preciso y fluido al estilo de Super Meat Boy se han ajustado manualmente parámetros como la velocidad, la aceleración y la fuerza del salto.

El nodo hijo **CollisionShape2D** define la hitbox del personaje, es decir, el área que el motor utiliza para detectar contactos con el entorno. El nodo **AnimatedSprite2D** gestiona las animaciones del sprite según el estado actual del personaje.

2.4.2 Sistema de colisiones y trampas

Las trampas (pinchos, sierras) son nodos independientes que contienen una **CollisionShape2D** configurada como zona de peligro. Cuando el personaje entra en contacto con ellas, se emite una **Signal** (señal) que el gestor del nivel recibe para ejecutar la muerte del jugador y redirigir a la pantalla de derrota.

Este sistema de señales es clave para mantener el código limpio y desacoplado: la trampa no necesita saber nada del personaje, simplemente avisa de que ha habido contacto, y es el gestor quien decide qué hacer.

Para las superficies jugables (suelo, plataformas y paredes) se utilizan nodos **StaticBody2D** con sus correspondientes formas de colisión, que el motor de físicas de Godot procesa automáticamente cada frame.

2.4.3 Sistema de menús (Control + VBoxContainer)

Los menús del juego (inicio y pausa) están construidos con nodos de interfaz de usuario de Godot, concretamente usando **Control** como nodo raíz y un **VBoxContainer** para organizar verticalmente los botones.

Cada botón tiene una señal conectada a un script GDScript que ejecuta la acción correspondiente: iniciar el juego (cargar la escena del nivel 1), reintentar o salir de la aplicación. Los elementos visuales de fondo se gestionan con nodos **TextureRect** y **ColorRect**.

2.4.4 Sistema de reinicio al morir

Cuando el personaje contacta con una trampa, no existe una pantalla de muerte intermedia. En su lugar, el nivel se reinicia instantáneamente. Esto se consigue mediante una **Signal** emitida por la trampa que el script del personaje recibe, ejecutando get_tree().reload_current_scene() para recargar la escena actual de forma inmediata.

Esta decisión de diseño es intencionada y está directamente inspirada en Super Meat Boy: el reinicio instantáneo elimina la fricción entre la muerte y el siguiente intento, manteniendo el ritmo y la tensión del juego sin penalizar al jugador con pantallas de carga o menús intermedios.

2.4.5 Shader de efecto retro (CanvasLayer + ColorRect)

Para conseguir la estética retro característica del juego se ha implementado un shader de efecto VHS/CRT descargado de la tienda oficial de assets de Godot Engine y modificado para adaptarlo al estilo visual del proyecto.

Técnicamente, el shader se aplica sobre un nodo **ColorRect** hijo de un **CanvasLayer**, lo que hace que el efecto se renderice por encima de toda la escena sin afectar a la lógica del juego. Las modificaciones realizadas sobre el código original del shader incluyen la adición de un efecto de ojo de pez, el aumento del contraste y el incremento de la frecuencia de la estática por segundo.

2.4.6 Sistema de audio (AudioStreamPlayer)

La música y los efectos de sonido se gestionan mediante nodos **AudioStreamPlayer** integrados en cada escena. Las pistas musicales han sido generadas con **Suno AI** y exportadas en formato compatible con Godot.

Cada menú y cada nivel cuenta con su propio nodo de audio, que se activa automáticamente al cargar la escena. Esto permite que cada planta de la torre tenga una ambientación sonora diferenciada, reforzando la identidad visual y narrativa de cada zona.

2.5 Definición de las tareas

En este apartado se detalla cómo se han llevado a cabo las tareas de investigación técnica descritas anteriormente, especificando qué se quería comprobar, qué componentes se utilizaron, el proceso seguido y las conclusiones obtenidas.

2.5.1 Prueba 1

Qué se quería comprobar: Verificar que el personaje respondiera de forma inmediata y precisa a las entradas del teclado, sin retrasos perceptibles y con una física de salto satisfactoria, comparable a la de los referentes del género.

Componentes utilizados: CharacterBody2D, CollisionShape2D, AnimatedSprite2D, script de movimiento en GDScript.

Proceso: Se creó una escena de prueba con plataformas básicas y se implementaron distintas versiones del script de movimiento, ajustando iterativamente los valores de velocidad, aceleración, gravedad y fuerza del salto. Se jugaron sesiones cortas comparando la respuesta del personaje con la de Super Meat Boy como referente principal.

Figura 14. Movimiento del Personaje

Conclusiones: Se determinaron los valores óptimos para cada parámetro de movimiento. Se comprobó que CharacterBody2D es más adecuado que RigidBody2D para este tipo de juego, ya que permite un control manual total sobre la física sin depender del motor de físicas automático de Godot, que resultaba demasiado impredecible para un plataformas de precisión.

2.5.2 Prueba 2

Qué se quería comprobar: Verificar que el personaje interactuara correctamente con todos los elementos del mapa: apoyarse en el suelo, chocar contra paredes, detectar pinchos y otros obstáculos sin errores de detección.

Componentes utilizados: TileMap, StaticBody2D, CollisionShape2D, Area2D, señales de Godot.

Proceso: Se diseñó un nivel de prueba con distintos tipos de superficies y trampas. Se probaron diferentes configuraciones de hitbox para el personaje y para los obstáculos, buscando el equilibrio entre una detección precisa y una experiencia que no resultara injusta para el jugador. Se verificó que las señales emitidas por las trampas desencadenaran correctamente el reinicio del nivel.

Figura 15. Colisiones del Personaje

Conclusiones: Se comprobó que usar Area2D para las trampas y StaticBody2D para las superficies transitables es la combinación más eficiente. Se ajustaron las hitboxes para que fueran ligeramente más pequeñas que el sprite visual del personaje, reduciendo la sensación de muertes injustas.

2.5.3 Prueba 3

Qué se quería comprobar: Validar que separar cada nivel en una escena independiente de Godot era la estrategia correcta para el proyecto, tanto a nivel de estabilidad técnica como de flujo de trabajo en equipo.

Componentes utilizados: Sistema de escenas de Godot, SceneTree

Proceso: Se crearon varias escenas de nivel por separado y se implementó la transición entre ellas. Se comprobó que modificar o romper una escena no afectaba a las demás, y que el flujo de trabajo colaborativo con GitHub era más fluido al trabajar en ficheros de escena independientes.

Figura 16. Pasar de Escena y Reiniciar

Conclusiones: La arquitectura basada en escenas independientes demostró ser la opción correcta. Facilitó el desarrollo paralelo entre los dos miembros del equipo y redujo los conflictos al subir cambios al repositorio. Además, el reinicio del nivel al morir se implementa de forma limpia, sin necesidad de lógica adicional.

2.5.4 Prueba 4

Qué se quería comprobar: Verificar que el shader de efecto VHS/CRT descargado de la tienda oficial de Godot se podía integrar correctamente y modificar para adaptarlo a la estética del juego sin afectar al rendimiento.

Componentes utilizados: CanvasLayer, ColorRect, shader GLSL descargado de la Asset Library de Godot.

Proceso: Se añadió el CanvasLayer con el ColorRect al que se le aplicó el shader. Se realizaron modificaciones sobre el código del shader para ajustar el efecto de ojo de pez, aumentar el contraste y la frecuencia de la estática. Se comprobó el rendimiento antes y después de aplicarlo, verificando que se mantuvieran los 60 FPS.

Figura 17. Implementación Shader

Conclusiones: El shader se integró sin problemas y no causó impacto apreciable en el rendimiento. Las modificaciones realizadas reforzaron la identidad visual retro del juego, aportando una atmósfera más inmersiva sin añadir complejidad técnica significativa al proyecto.

2.5.5 Prueba 5

Qué se quería comprobar: Verificar que las pistas musicales generadas con Suno AI podían importarse y reproducirse correctamente en Godot, y que el cambio de música entre escenas funcionaba sin interrupciones ni solapamientos.

Componentes utilizados: AudioStreamPlayer, archivos de audio exportados desde Suno AI.

Proceso: Se exportaron las pistas desde Suno en formato compatible con Godot y se añadieron como recursos de audio. Se colocó un nodo AudioStreamPlayer en cada escena configurado para reproducirse automáticamente al cargar el nivel. Se comprobó que al cambiar de escena el audio de la anterior se detuviera correctamente.

Figura 18. Efectos de Sonido

Conclusiones: La integración fue directa y sin problemas técnicos. Godot gestiona bien la carga y reproducción de audio, y el hecho de tener el AudioStreamPlayer dentro de cada escena garantiza que la música se detenga automáticamente al hacer la transición, sin necesidad de lógica adicional.

2.6 Definición de las funcionalidades

A continuación se describen todas las funcionalidades que ofrece Ascend Tower, detallando cómo está implementada cada una y su estado actual de desarrollo.

2.6.1 Funcionalidad 1 Movimiento del personaje

Qué permite hacer: El jugador puede mover al personaje lateralmente, saltar, hacer wall jump y beneficiarse del coyote time para un control preciso y fluido.

Cómo funciona: Un script en GDScript procesa las entradas del teclado en cada frame mediante `_physics_process(delta)`. El nodo CharacterBody2D gestiona el movimiento y las colisiones con el entorno. El coyote time se implementa con un contador de frames que se activa al dejar de tocar el suelo, permitiendo saltar durante un breve margen. El wall jump detecta el contacto con paredes y permite al personaje impulsarse en dirección contraria al pulsar el salto.

Estado: Totalmente implementado.

Figura 19. Funcionalidad 1

2.6.2 Funcionalidad 2 Sistema de trampas y reinicio

Qué permite hacer: El jugador puede interactuar con todos los elementos del mapa, incluyendo trampas como pinchos y sierras. Al contactar con una trampa el nivel se reinicia instantáneamente sin pantalla intermedia.

Cómo funciona: Las trampas son nodos con un Area2D que detectan la entrada del personaje en su zona de colisión. Al producirse el contacto, se emite una señal que ejecuta `reload_current_scene()`, recargando el nivel actual de forma inmediata. Esta decisión de diseño elimina la fricción entre la muerte y el siguiente intento, manteniendo el ritmo del juego.

Estado: Totalmente implementado.

Figura 20. Funcionalidad 2

2.6.3 Funcionalidad 3 Modo Extremo

Qué permite hacer: El jugador puede acceder a una versión de alta dificultad de cada nivel desde el menú principal, con una densidad de obstáculos significativamente mayor.

Cómo funciona: Cada nivel extremo es una escena independiente que replica la estructura del nivel normal pero con más trampas y menor margen de error. Se accede desde el menú principal mediante un botón que carga directamente la secuencia de niveles extremos, siguiendo la misma lógica de transición entre escenas que el modo normal.

Estado: Totalmente implementado

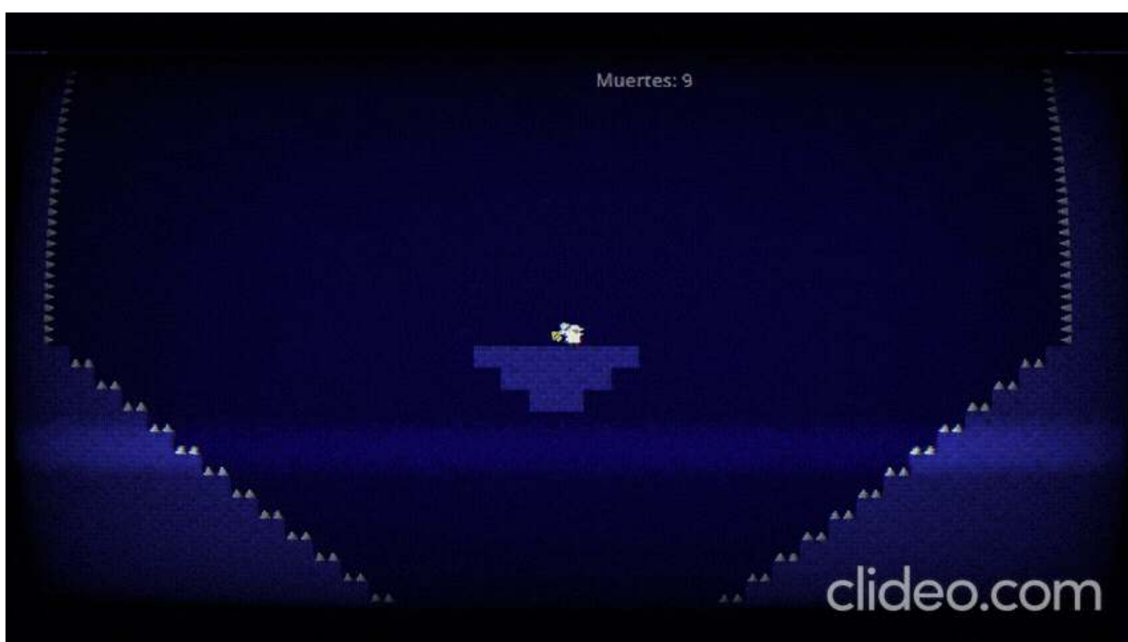


Figura 21. Funcionalidad 3

2.6.4 Funcionalidad 4 Sistema de menús

Qué permite hacer: El jugador puede iniciar el juego en modo normal o extremo, y salir de la aplicación desde el menú principal.

Cómo funciona: Los menús están construidos con nodos Control y VBoxContainer. Cada botón tiene una señal conectada a un script GDScript que ejecuta la acción correspondiente: cargar la escena del nivel 1, cargar la secuencia de niveles extremos o cerrar la aplicación con `get_tree().quit()`. Los fondos se gestionan con TextureRect y ColorRect.

Estado: Totalmente implementado.

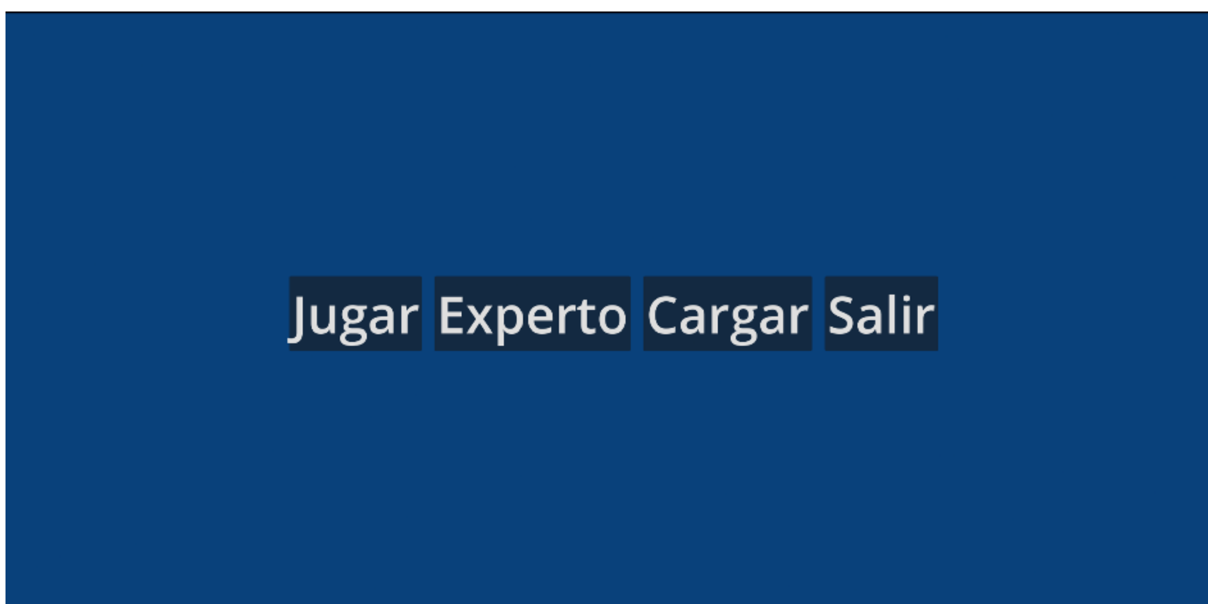


Figura 22. Funcionalidad 4

2.6.5 Funcionalidad 5 Música

Qué permite hacer: El juego reproduce música original de fondo en cada nivel y menú, cambiando automáticamente al transicionar entre escenas.

Cómo funciona: Cada escena contiene un nodo AudioStreamPlayer configurado para reproducirse automáticamente al cargarse. Las pistas han sido generadas con Suno AI y exportadas en formato compatible con Godot. Al cambiar de escena, el AudioStreamPlayer de la escena anterior se destruye junto con ella, evitando solapamientos de audio.

Estado: Totalmente implementado.

Figura 23. Funcionalidad 5

2.6.6 Funcionalidad 6 Guardar y cargar partida

Qué permite hacer: El jugador puede guardar su progreso en el modo normal y retomarlo más tarde desde el último nivel que había alcanzado, sin necesidad de empezar desde el principio.

Cómo funciona: Al cerrar el juego, el sistema guarda automáticamente el nivel actual del jugador. Al volver a iniciar la aplicación, el jugador tiene la opción de pulsar "Cargar" en el menú principal en lugar de "Jugar", lo que carga directamente la escena del último nivel guardado. Esta funcionalidad solo está disponible en el modo normal, ya que el Modo Extremo está pensado para jugadores que ya dominan el juego completo.

Estado: Totalmente implementado.

Figura 24. Funcionalidad 6

3 Otros capítulos

3.1 Por qué hicimos este proyecto?

La motivación principal detrás de este proyecto es que ambos miembros del equipo llevamos años jugando videojuegos y siempre habíamos querido intentar crear uno por nuestra cuenta. Este proyecto representaba la oportunidad perfecta para transformar esa idea en algo real.

La decisión de hacer un juego 2D en lugar de 3D fue una decisión técnica.Inicialmente barajamos un proyecto en tres dimensiones, pero tras analizar el tiempo disponible y nuestro nivel de experiencia con motores de videojuegos que era nulo al inicio determinamos que un proyecto 3D implicaba demasiados frentes abiertos simultáneamente: animación, iluminación y física 3D. Optar por el 2D nos permitió centrar todo el esfuerzo en lo que realmente define la calidad de este género: el movimiento del personaje y el diseño de niveles.

La elección de Godot como motor también fue deliberada. Desde el principio ya sabíamos que lo haríamos en Godot ya que ofrece tres ventajas decisivas para nuestro caso: es completamente gratuito, está especialmente optimizado para juegos 2D y no impone restricciones de licencia que pudieran complicar la distribución del juego. Además, su lenguaje nativo GDScript tiene una sintaxis más intuitiva que otro motor lo que facilitó enormemente el aprendizaje desde cero.

3.2 Ideas

El proyecto pasó por tres conceptos distintos antes de llegar a su forma final, y cada cambio estuvo justificado por criterios técnicos y de viabilidad.

Primera idea: FPS 3D multijugador online La propuesta inicial era un juego de disparos en primera persona inspirado en Team Fortress 2, con selección de personajes y partidas online. Se descartó por dos motivos principales: la complejidad del modelado y animación en 3D para un equipo sin experiencia previa, y la dificultad técnica de implementar un sistema multijugador online en el tiempo disponible. Hubiera sido un proyecto inacabable.

Figura 25. Idea

Figura 26. Idea 2

Figura 27. Selección personaje

Segunda idea: Juego de terror 3D Se planteó brevemente un juego de terror en primera persona inspirado en Slenderman. Se descartó por falta de originalidad y porque seguía arrastrando el problema del modelado 3D, que era el obstáculo principal independientemente del género.

Figura 28. Idea de juego de terror

Figura 29. Idea juego terror 2

Figura 30. Idea de juego de terror 3

Tercera idea: Metroidvania 2D Durante la preproducción del proyecto actual se exploró un formato Metroidvania inspirado en Hollow Knight, Silksong y Nine sols, con mapas abiertos de exploración. Se detectó que el moveset rápido y preciso que queríamos implementar no encajaba bien con mapas grandes y no lineales el jugador necesitaba orientación constante y los combates con enemigos añadían una capa de complejidad que desviaba el foco del movimiento. Esta fase no fue un fracaso sino un aprendizaje que llevó directamente al concepto final.

Figura 31. Hollow knight

Figura 32. Hollow knight silksong

Figura 33. Nine sols

Concepto final: Plataformas de precisión por niveles La solución fue hacer un plataformas de niveles definidos donde la mecánica principal es el movimiento del personaje, siguiendo el modelo de Super Meat Boy y Celeste. Este formato permite niveles cortos e intensos donde cada muerte es responsabilidad del jugador y no del diseño, lo que encaja perfectamente con el moveset preciso que queríamos ofrecer. La ambientación retro, inspirada en clásicos arcade como Castlevania y Donkey Kong, que mejoró la identidad visual del proyecto.

Figura 34. Super meat boy

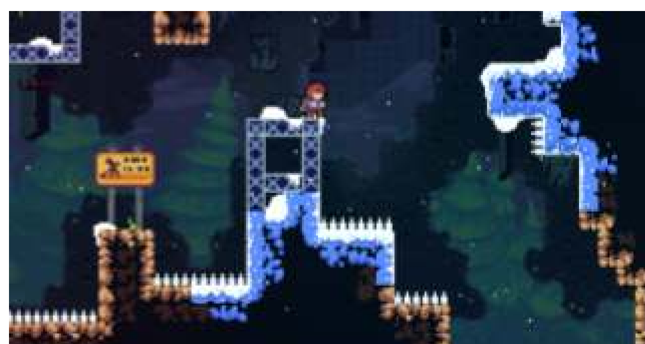


Figura 35. Celeste

Figura 36. Donkey Kong

Figura 37. Castlevania

3.3 Juego en general

Ascend Tower está ambientado en una antigua torre misteriosa que, según la leyenda, esconde en su interior incontables tesoros. El protagonista, empujado por la pobreza, decide arriesgarlo todo y adentrarse en ella con un único objetivo, recorrer todas sus plantas, recoger los tesoros que guarda cada una y salir con vida para volverse rico.

El objetivo de cada nivel es avanzar a la siguiente planta de la torre recogiendo el tesoro que la desbloquea, todo ello sorteando pinchos, sierras y demás obstáculos que protegen el camino. El diseño del movimiento está directamente inspirado en Super Meat Boy, con saltos amplios y fluidos, control de velocidad en el aire y una respuesta inmediata a los controles. La estructura de ascender planta a planta es una inspiración parcial de Celeste, donde el jugador debe superar sus propios límites para llegar a la cima.

Para los jugadores que buscan un reto mayor, cada nivel cuenta con una versión de Modo Extremo donde el margen de error es prácticamente nulo y la densidad de obstáculos hace que incluso los jugadores más experimentados tengan que concentrarse al máximo.

Figura 38. Juego en general

3.4 Godot

3.4.1 Primeros diseños de niveles

Los primeros días del desarrollo los dedicamos a familiarizarnos con Godot antes de construir nada definitivo. Aunque es uno de los motores más accesibles del mercado, partir desde cero implica entender conceptos fundamentales como el sistema de escenas, la jerarquía de nodos y el sistema de señales, sin los cuales es imposible estructurar un proyecto correctamente.

Esta inversión de tiempo inicial fue una decisión deliberada, preferimos dedicar los primeros días a entender la lógica del motor que empezar a construir niveles sin una base sólida y tener que rehacer el trabajo más adelante.

Una vez asimilados los conceptos básicos, el primer paso fue buscar un paquete de assets en Itch.io para las texturas del juego. Se valoraron varias opciones gratuitas y se eligió un paquete que cubría tanto los elementos del entorno (plataformas, fondos, decoración) como el sprite del personaje jugable, con una estética pixel art coherente con la ambientación retro que buscábamos.

Con estos assets se construyó un nivel de prueba cuyo único objetivo era verificar que el movimiento del personaje y las hitboxes funcionaban correctamente antes de diseñar los niveles definitivos. Se detectaron y corrigieron varios problemas de detección de colisiones en esta fase, lo que evitó que esos errores se propagaran a todos los niveles del juego.

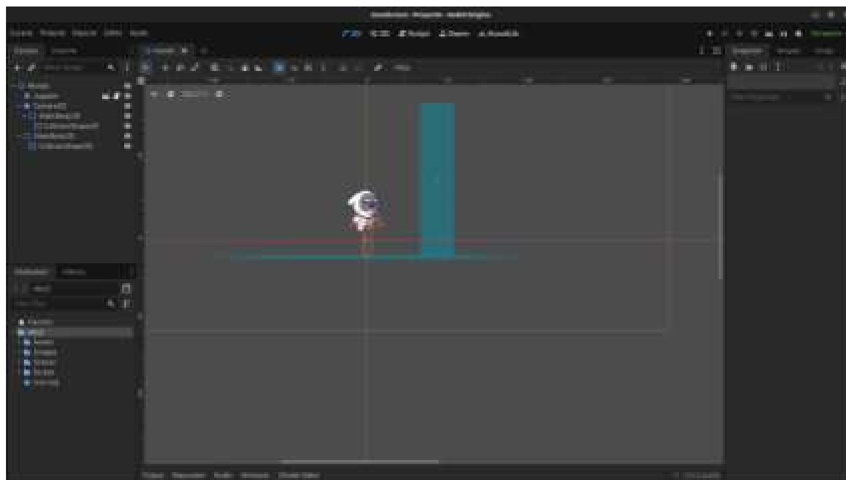


Figura 39. Probando las hitbox

Como se muestra en la captura, durante esta fase inicial estábamos experimentando con las hitboxes del personaje y los elementos del entorno. Una hitbox es un área de colisión invisible que rodea a un objeto y define cómo interactúa físicamente con el mundo: puede actuar como una superficie sólida, impidiendo el paso del personaje, o como una zona de peligro que provoca el reinicio del nivel al contactar con él. Ajustar correctamente estas áreas es fundamental en un plataformas de precisión, ya que una hitbox demasiado grande genera muertes injustas y una demasiado pequeña hace que el juego pierda coherencia visual.

Este nivel de prueba tenía como único objetivo validar dos aspectos críticos antes de empezar a construir los niveles definitivos: que el movimiento del personaje respondiera de forma correcta y fluida, y que las hitboxes de los bloques y obstáculos detectaran las colisiones sin errores. Una vez identificados y corregidos todos los problemas detectados en esta fase, tanto los relacionados con el comportamiento del personaje como los de detección de colisiones, se dio por cerrada la etapa de prototipado y se inició el diseño de los niveles definitivos del juego.

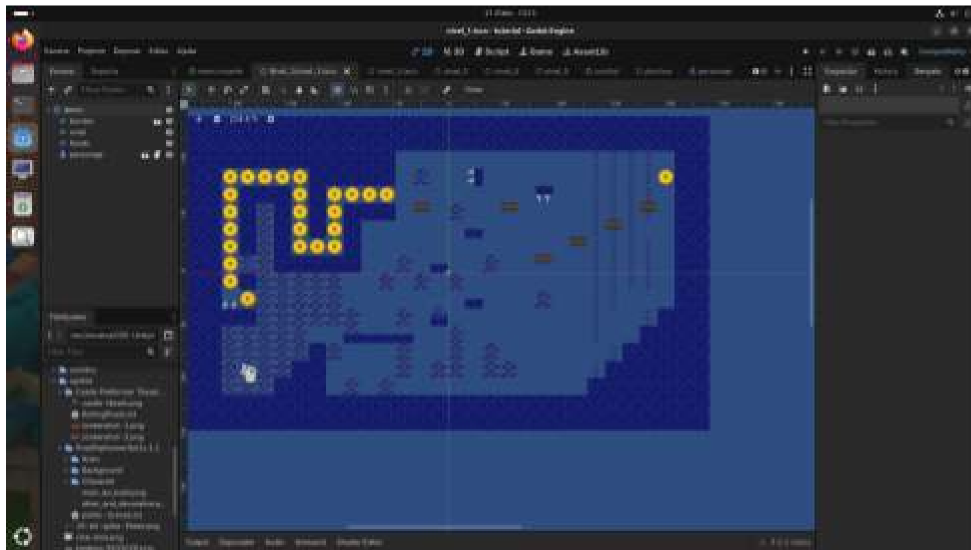


Figura 40. Ejemplo de mapa inicial

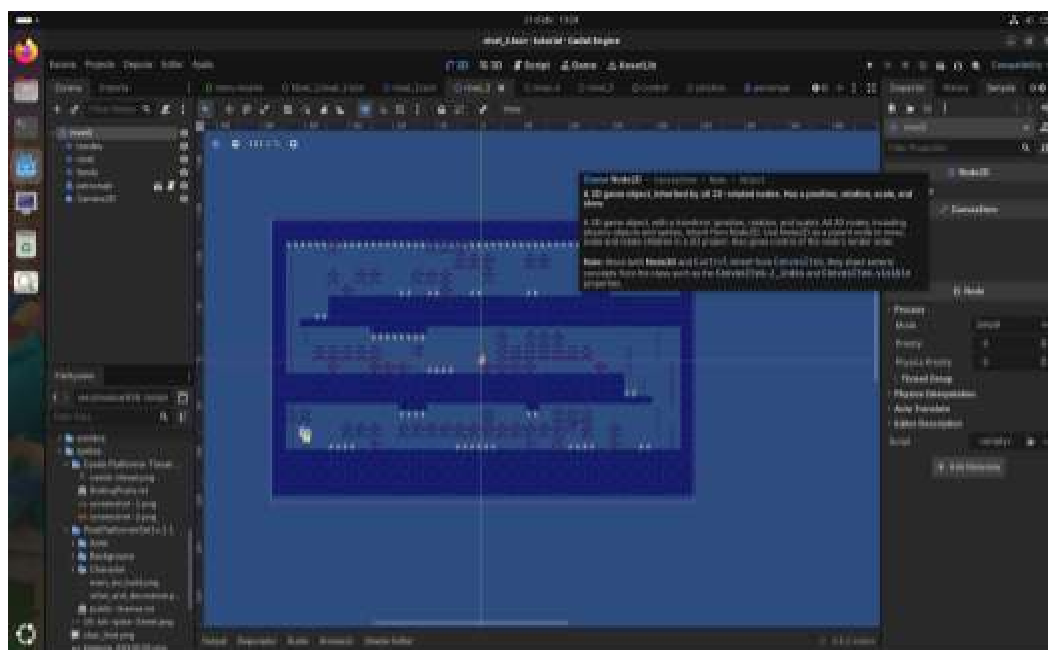


Figura 41. Ejemplo de mapa inicial 2

3.4.2 Shader

Uno de los elementos más característicos de Ascend Tower es su efecto visual retro, que simula la apariencia de un monitor CRT antiguo con estática y distorsión. Este efecto aporta coherencia estética con las referencias arcade del proyecto y refuerza la inmersión.

Para implementarlo se valoraron dos alternativas: crear un shader propio desde cero o utilizar uno ya existente como punto de partida. Dado que ninguno de los dos miembros del equipo tenía experiencia previa con programación de shaders en GLSL, optar por uno propio hubiera supuesto un coste de tiempo desproporcionado para el resultado. Se optó por descargar el shader VHS/CRT de A7meD disponible en la Asset Library oficial de Godot y modificarlo para adaptarlo a las necesidades del proyecto.

Las modificaciones realizadas fueron tres: se añadió un efecto de ojo de pez para acentuar la curvatura de pantalla, se aumentó el contraste para reforzar los colores del pixel art, y se incrementó la frecuencia de la estática para hacerla más visible. Técnicamente, el shader se aplica sobre un nodo ColorRect hijo de un CanvasLayer, lo que garantiza que el efecto se renderice por encima de toda la escena sin interferir con la lógica del juego ni afectar al rendimiento.

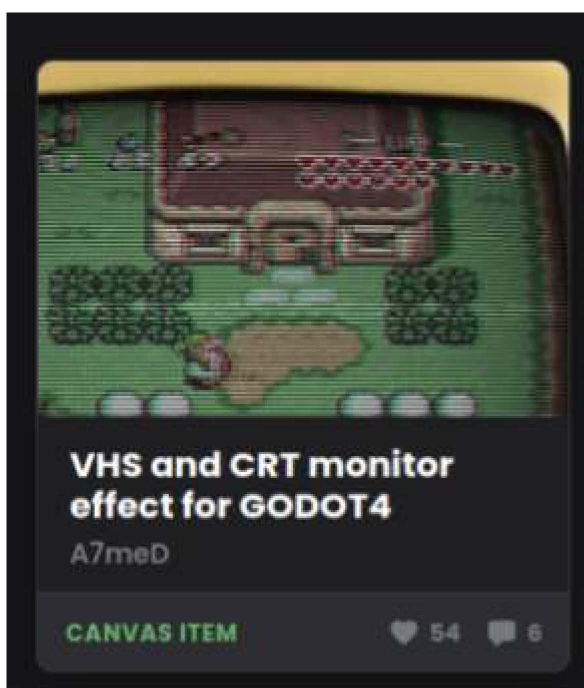


Figura 42. Shader de Godot

3.4.3 Programación de menús

El juego requería dos menús principales: el menú de inicio y un menú de reinicio accesible al morir. Para implementarlos se consultaron tutoriales de YouTube específicos de Godot, ya que la documentación oficial resultaba demasiado genérica para un caso de uso tan concreto.

Ambos menús siguen la misma estructura de nodos: un nodo Control como raíz, un VBoxContainer para organizar los botones verticalmente, nodos TextureRect y ColorRect para los elementos visuales de fondo, y un AudioStreamPlayer para la música. Cada botón tiene una señal conectada al script del menú que ejecuta la acción correspondiente.

El menú de inicio ofrece las opciones de jugar en modo normal, jugar en Modo Extremo, cargar partida guardada y salir. El menú de reinicio aparece al pausar el juego y permite al jugador volver al nivel 1 o salir de la aplicación. Es importante destacar que cuando el jugador toca una trampa no aparece ningún menú intermedio el nivel se reinicia instantáneamente mediante `reload_current_scene()`, una decisión de diseño inspirada en Super Meat Boy que elimina la fricción entre la muerte y el siguiente intento.

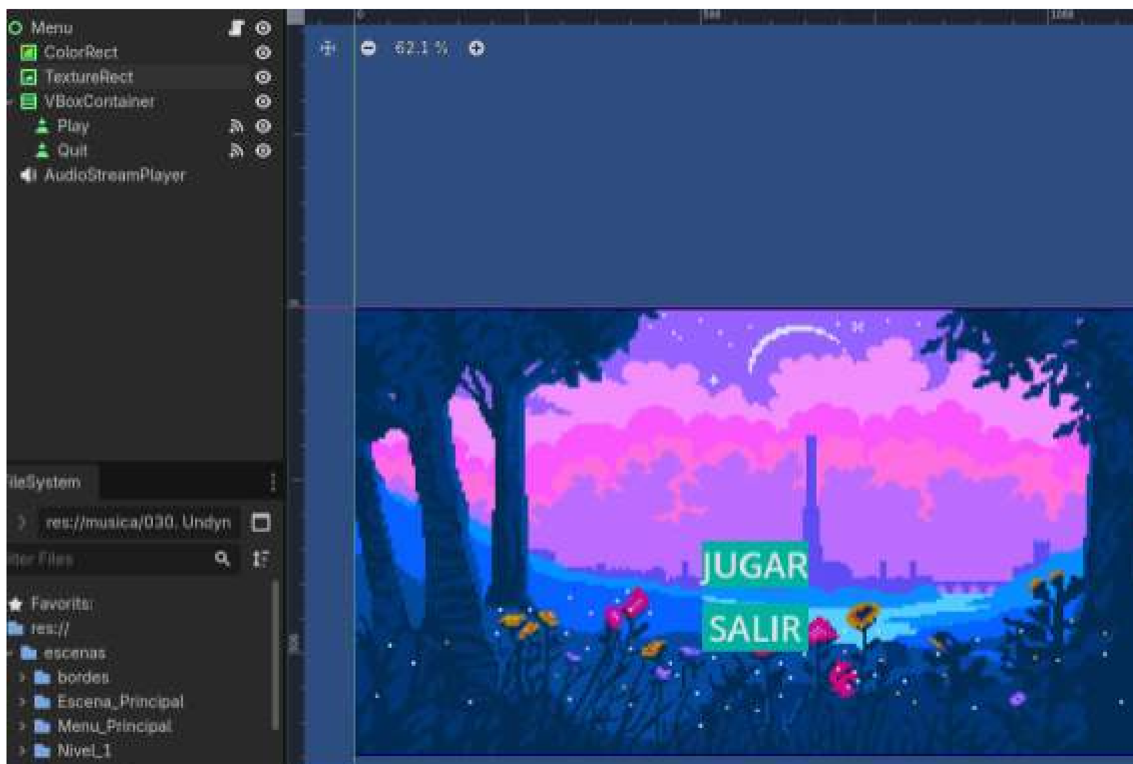


Figura 43. Menú inicio prototipo

3.4.4 Niveles extremos.

El Modo Extremo nació de la idea sobre la rejugabilidad del proyecto. Un juego de plataformas de precisión tiene una vida útil limitada una vez que el jugador domina los niveles normales, por lo que se decidió añadir una variante de alta dificultad para algunas plantas de la torre.

Cada nivel extremo es una escena independiente que replica la estructura del nivel normal pero con una densidad de obstáculos significativamente mayor y un margen de error prácticamente nulo. La decisión de hacerlos como escenas separadas en lugar de modificar los niveles originales mediante parámetros fue técnica: mantener los niveles normales intactos garantiza que un cambio en el Modo Extremo no pueda romper accidentalmente la experiencia principal del juego.

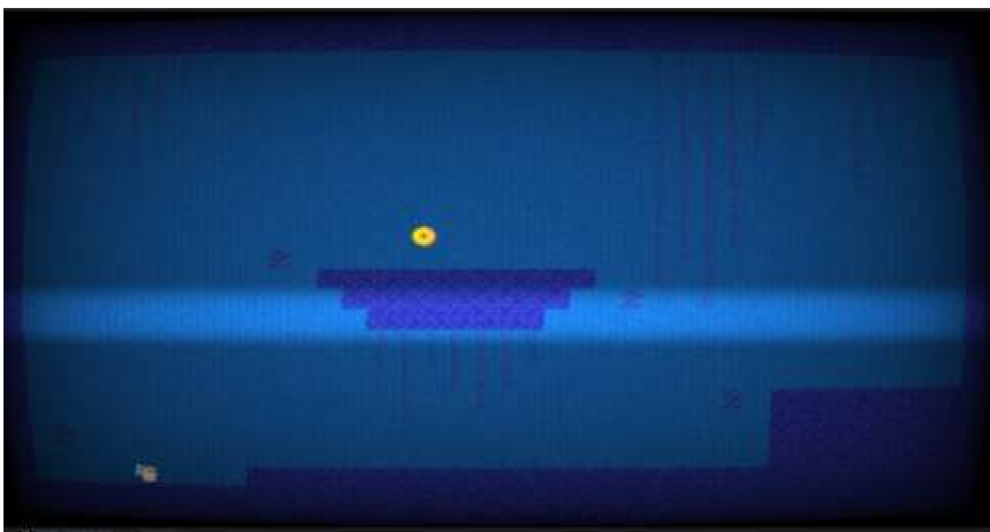


Figura 44. Nivel normal

Figura 45. Nivel extremo.

3.5 GitHub

Para gestionar el trabajo colaborativo entre los dos miembros del equipo se utilizó GitHub como sistema de control de versiones. Se valoró la alternativa de compartir archivos mediante Google Drive o correo electrónico, pero se descartó porque intercambiar archivos del proyecto de Godot manualmente genera fácilmente conflictos y pérdidas de trabajo cuando dos personas modifican los mismos ficheros.

Se crearon dos repositorios diferenciados: uno para los prototipos y scripts en desarrollo, y otro para el código de la página web. El flujo de trabajo consistía en subir los avances semanalmente al repositorio compartido mediante commits, de forma que el otro miembro del equipo pudiera descargar la versión más reciente con `git clone` al inicio de cada sesión de trabajo. Este sistema garantizó que ningún avance se perdiera y que los conflictos entre versiones fueran mínimos.

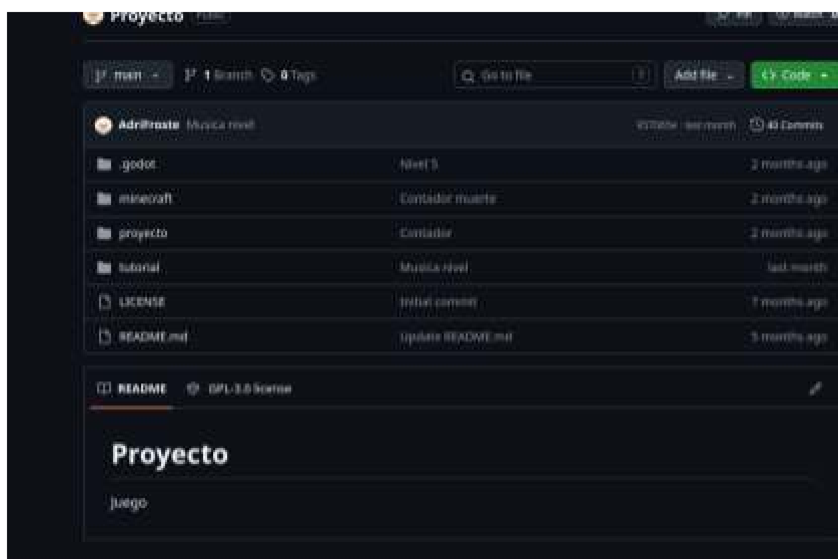


Figura 46. Primer repositorio de pruebas para Github

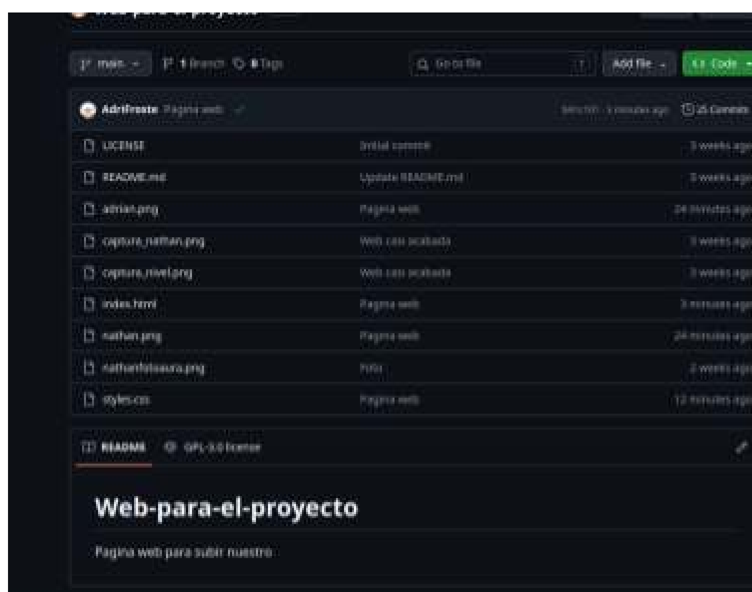


Figura 47. Repositorio para la web.

3.6 Web

Para cumplir con el objetivo de publicación del proyecto se desarrolló una página web propia alojada en GitHub Pages. Se valoró la alternativa de publicar el juego únicamente en Itch.io, pero se decidió crear una web propia para tener un espacio con identidad propia donde presentar el proyecto de forma más completa y profesional.

La web está desarrollada con HTML y CSS, organizados en un archivo index.html y un styles.css, con imágenes del juego a lo largo de la página. Su objetivo es presentar el proyecto a cualquier usuario que no lo conozca, ofreciéndole información sobre la historia y las mecánicas, capturas del juego y un enlace de descarga del ejecutable.

Uno de los elementos más destacados de la web es la sección de sugerencias y reseñas, implementada mediante Formspree. Se valoraron alternativas como crear un backend propio o usar Google Forms, pero Formspree resultó la opción más eficiente: permite añadir un formulario de contacto directamente en el HTML sin necesidad de servidor propio, y los mensajes llegan automáticamente al correo del equipo. El usuario introduce su nombre y su opinión, y el mensaje se envía directamente a la bandeja de entrada sin necesidad de ninguna configuración adicional por parte del jugador.

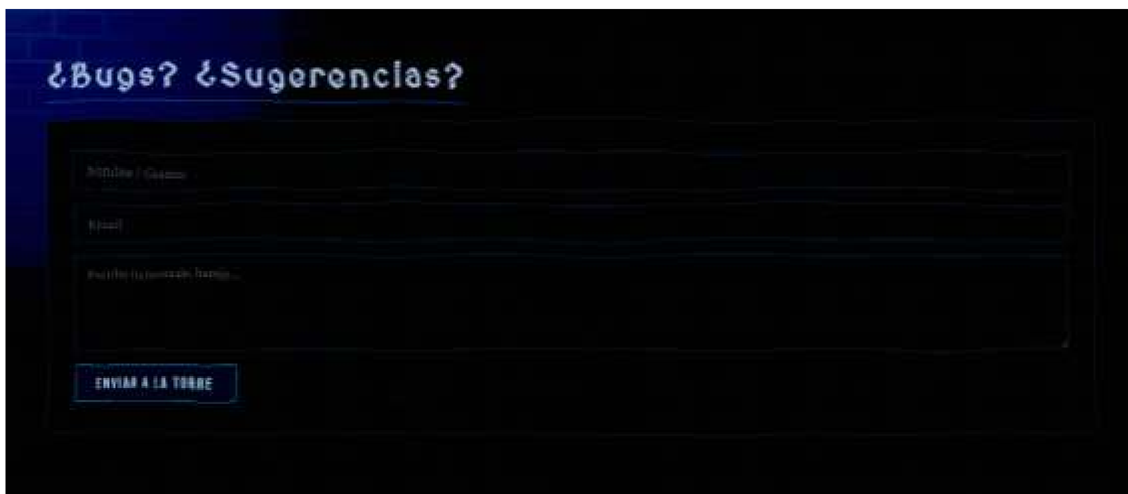


Figura 48. Buzón de sugerencias

Figura 49. Página web

4 Conclusiones

4.1 Conclusiones generales del proyecto

Desarrollar este videojuego nos ha permitido comprender de primera mano todo el trabajo que hay detrás de una mecánica o un nivel que como jugadores damos por hecha. Hemos pasado de ser usuarios a ser creadores, y ese cambio de perspectiva ha hecho que apreciemos más a las personas que dedican años de su vida para realizar un producto

A nivel académico, enfrentarnos a errores, bugs y comportamientos inesperados del código nos ha obligado a desarrollar una capacidad de resolución de problemas real, buscar información de forma autónoma, leer documentación técnica en inglés y no abandonar cuando algo no funcionaba como se esperaba. También aprendimos a gestionar el alcance de un proyecto, reduciendo las ideas iniciales a algo realista y terminable dentro del tiempo disponible.

A nivel de trabajo en equipo, coordinar el desarrollo con GitHub, repartirnos las tareas y evitar conflictos entre versiones ha sido una experiencia muy cercana a lo que es trabajar en un equipo de desarrollo profesional real. En cuanto a lo técnico, nos llevamos una base sólida de Godot Engine y GDScript que sin duda será útil más allá de este proyecto. En resumen, hemos entregado un juego funcional, visualmente coherente y, sobre todo, completamente desarrollado por nosotros desde cero.

4.2 Consecución de los objetivos

En relación a los objetivos planteados al inicio del proyecto, podemos considerar que la gran mayoría se han alcanzado satisfactoriamente.

Desarrollar un videojuego de plataformas 2D completo Cumplido. El juego cuenta con mecánicas de movimiento fluidas, entre 5 y 8 niveles funcionales con dificultad progresiva y el Modo Extremo implementado en todos ellos, ofreciendo una experiencia de juego sólida y completa.

Capacitación en el motor de juegos Cumplido. Partiendo desde un conocimiento inicial nulo, hemos sido capaces de programar de forma autónoma físicas, colisiones, shaders, menús, sistema de audio y guardado de partida dentro de Godot.

Aprender Godot y programar las mecánicas principales Cumplido. Se han implementado mecánicas avanzadas como el wall jump y el coyote time, además de toda la lógica de colisiones, señales y transiciones entre escenas.

Lograr un control fluido del personaje Cumplido. Tras varias iteraciones ajustando los parámetros de velocidad, aceleración, gravedad y salto, se obtuvo un movimiento que cumple con los estándares del género.

Crear niveles con dificultad progresiva Cumplido. Los niveles están diseñados para introducir las mecánicas gradualmente y aumentar el reto a medida que el jugador avanza por la torre.

Desarrollar un Modo Extremo Cumplido. juego cuenta con niveles de alta dificultad para aumentar la rejugabilidad, accesible desde el menú principal.

Implementar una plataforma de distribución Cumplido. Se ha desarrollado una página web propia con el juego disponible para descarga directa, y el juego está publicado adicionalmente en Itch.io como plataforma de distribución secundaria.

4.3 Valoración de la metodología y planificación

La metodología ágil basada en Scrum demostró ser la más adecuada para este proyecto. Su capacidad de adaptación fue clave, ya que el concepto del juego pasó por tres versiones distintas — FPS 3D, Metroidvania y plataformas 2D — antes de llegar a su forma final. Con una metodología tradicional en cascada, cada uno de esos cambios hubiera supuesto reiniciar toda la planificación desde cero, algo inviable en el tiempo disponible.

El uso de Trello para organizar las tareas y GitHub para el control de versiones nos permitió coordinarnos de forma efectiva. Sin embargo, reconocemos que al principio no aprovechamos estas herramientas todo lo que deberíamos, lo que provocó algunos momentos de desorganización en las primeras semanas.

Respecto a los plazos, se cumplieron en términos generales, aunque la curva de aprendizaje de Godot fue más pronunciada de lo esperado al partir desde cero. Esto hizo que las primeras semanas fueran menos productivas de lo previsto. En retrospectiva, haber invertido más tiempo inicial en aprender el motor antes de empezar a desarrollar niveles habría sido más eficiente, aunque este aprendizaje acelerado también fue en sí mismo uno de los objetivos del proyecto.

4.4 Visión a futuro

De cara al futuro, existen varias líneas de trabajo que no pudieron explorarse durante el desarrollo pero que serían interesantes para dar continuidad al proyecto.

En primer lugar, sería interesante **ampliar el contenido** añadiendo más plantas a la torre. La historia del protagonista que se adentra en busca de los tesoros que esconde tiene potencial narrativo para desarrollarse con elementos de lore, mensajes ambientales o cinemáticas entre niveles que enriquezcan la experiencia.

También sería valioso **implementar un sistema de puntuaciones global** donde los jugadores pudieran comparar sus mejores tiempos, añadiendo un componente competitivo que aumentaría significativamente la rejugabilidad del juego.

Otra línea de trabajo sería el **port a otras plataformas**. Godot permite exportar a Android y HTML5 de forma relativamente sencilla, lo que ampliaría el público al que puede llegar el juego sin necesidad de reescribir el código.

Por último, **desarrollar sprites propios** en lugar de utilizar assets descargados de Itch.io daría al juego una identidad visual más única y profesional, algo que por limitaciones de tiempo no fue posible abordar durante este desarrollo.

5. Glosario

Script: Es un conjunto de instrucciones o comandos escritos en un lenguaje de programación, diseñado para automatizar tareas, ejecutar acciones específicas o controlar otras aplicaciones. A diferencia de los programas compilados, suelen ser interpretados línea por línea por un motor o intérprete en tiempo

Nodo: es la unidad funcional más pequeña y básica para construir juegos. Actúa como un bloque de construcción predefinido con un propósito específico (mostrar imágenes, reproducir sonido, gestionar físicas, etc.) que se organiza en estructuras de árbol para formar escenas

Area2d: En Godot es un nodo especializado en la detección de superposiciones (overlap) en un espacio 2D. A diferencia de los nodos de física rígida (RigidBody2D o CharacterBody2D), el Area2D no detiene objetos ni causa colisiones físicas sólidas; en su lugar, se utiliza para detectar cuándo otros objetos de colisión (CollisionObject2D) entran, salen o están dentro de un área definida.

StaticBody2d: Es un cuerpo físico 2D diseñado para ser inamovible por fuerzas externas, colisiones o gravedad. Es el nodo base ideal para objetos del entorno que actúan como obstáculos sólidos, tales como suelos, paredes, plataformas fijas o techos.

Shader: Es un pequeño programa informático que se ejecuta en la GPU (tarjeta gráfica) para determinar cómo se dibuja cada píxel o vértice en pantalla, controlando luces, sombras, colores y texturas en tiempo real. Son esenciales en videojuegos y renderizado para crear efectos visuales como agua, fuego, metal o iluminación realista.

Sprites: Es una imagen o mapa de bits 2D que se integra en una escena más grande (generalmente 2D) para representar personajes, objetos, enemigos o efectos especiales. Funcionan como elementos independientes del fondo, permitiendo movimiento e interacción, y pueden animarse cambiando rápidamente entre distintos sprites.

Scenes: Es un concepto fundamental que define un grupo estructurado y reutilizable de nodos organizados en forma de árbol. Se pueden entender como bloques de construcción modulares que, al combinarse, forman todo el juego

TileSet: Es un recurso (un archivo `.tres` o integrado en una escena) que actúa como una biblioteca o colección de "tiles" (teselas o piezas pequeñas). Se utiliza específicamente en conjunto con el nodo TileMap o TileMapLayer para diseñar niveles, escenarios y fondos 2D de manera eficiente

TileMap: Es un nodo especializado en la creación de escenarios y mundos 2D basados en una cuadrícula (grid) de mosaicos o teselas (tiles). Es la herramienta principal para diseñar niveles de forma eficiente, permitiendo "pintar" el mapa de juego utilizando piezas pequeñas reutilizables.

Metroidvania: Es un subgénero de videojuegos de acción-aventura centrado en la exploración no lineal de un mapa interconectado, donde el progreso está bloqueado por habilidades que el jugador debe conseguir más adelante. El nombre proviene de la fusión de *Metroid* y *Castlevania*, pioneros en este estilo de juego, por ejemplo el proyecto de nuestro compañero Eric Aguilar es un Metroidvania

Pixel art: Es una forma de arte digital donde las imágenes se crean y editan a nivel de píxel individual, utilizando pequeños cuadrados de color como bloques básicos para construir formas, similar a un mosaico.

CharacterBody2D: Es un nodo de Godot Engine diseñado específicamente para representar personajes controlados por el jugador o por la inteligencia artificial. A diferencia de RigidBody2D, permite un control manual total sobre el movimiento y las físicas, sin depender del motor de físicas automático del motor. Es el nodo base ideal para personajes de plataformas de precisión donde se requiere una respuesta inmediata a los controles.

CollisionShape2D: Es un nodo hijo que define la forma geométrica utilizada por el motor de físicas de Godot para detectar colisiones. Puede tener distintas formas primitivas como rectángulos, círculos o cápsulas. Se utiliza en combinación con nodos como CharacterBody2D, StaticBody2D o Area2D para determinar el área física de un objeto en el mundo del juego.

GDScript: Es el lenguaje de programación nativo e integrado en Godot Engine. Se trata de un lenguaje de alto nivel, imperativo y orientado a objetos, con una sintaxis clara e inspirada en Python. Está optimizado específicamente para trabajar con el sistema de nodos de Godot, lo que permite un desarrollo más rápido y fluido en comparación con otros lenguajes compatibles con el motor como C#.

Signal (Señal): Es un sistema de comunicación entre nodos propio de Godot Engine, basado en el patrón de diseño observador. Permite que un nodo emita un aviso cuando ocurre un evento concreto (como una colisión o la pulsación de un botón) y que otros nodos suscritos reaccionen a ese aviso ejecutando una función. Su uso mantiene el código limpio y desacoplado, ya que los nodos no necesitan conocerse entre sí directamente.

Hitbox: Es un área de colisión invisible que rodea a un personaje u objeto en un videojuego y define cómo interactúa físicamente con el mundo. Puede actuar como una superficie sólida que impide el paso, o como una zona de peligro que desencadena una acción al ser tocada. En un plataformas de precisión, ajustar correctamente las hitboxes es fundamental, ya que una hitbox demasiado grande genera muertes injustas y una demasiado pequeña hace que el juego pierda coherencia visual.

FPS (Frames Per Second): Es la unidad de medida que indica cuántos fotogramas por segundo renderiza y muestra un videojuego. A mayor número de FPS, más fluida es la imagen en pantalla. En videojuegos de precisión, mantener una tasa estable de 60 FPS es un requisito técnico crítico, ya que cualquier bajada de rendimiento puede afectar directamente a la respuesta de los controles y perjudicar la experiencia del jugador.

Coyote Time: Es una técnica de diseño de juegos de plataformas que permite al jugador ejecutar un salto durante un breve margen de tiempo después de haber salido del borde de una plataforma, incluso cuando el personaje ya está en el aire. Su nombre proviene del personaje animado Coyote, que en los dibujos animados continúa corriendo un instante antes de caer al vacío. Esta mecánica reduce la sensación de muertes injustas y hace que el control del personaje se sienta más natural y fluido.

Wall Jump: Es una mecánica de movimiento presente en juegos de plataformas que permite al personaje impulsarse desde una pared vertical para ganar altura o cambiar de dirección. Se activa cuando el personaje está en contacto con una pared y el jugador pulsa el botón de salto, aplicando una fuerza en dirección contraria a la pared. Es una mecánica avanzada que añade profundidad al movimiento y permite diseñar niveles con retos verticales más complejos.

Physics Engine (Motor de físicas): Es el sistema interno de un motor de videojuegos encargado de simular el comportamiento físico de los objetos en el mundo del juego, incluyendo la gravedad, las colisiones, el rebote y la fricción. En Godot Engine, el motor de físicas procesa automáticamente cada frame las interacciones entre los distintos cuerpos físicos de la escena, como plataformas, personajes y trampas.

GPU (Graphics Processing Unit): Es el procesador gráfico de un ordenador, diseñado específicamente para realizar cálculos matemáticos en paralelo a gran velocidad. En el contexto de los videojuegos, es la encargada de renderizar cada fotograma en pantalla, ejecutar los shaders y procesar los efectos visuales en tiempo real. A diferencia de la CPU, que está optimizada para tareas secuenciales, la GPU puede procesar miles de píxeles simultáneamente.

Motor de juego (Game Engine): Es un framework o entorno de desarrollo software que proporciona las herramientas, sistemas y bibliotecas necesarios para crear videojuegos. Incluye funcionalidades como el renderizado gráfico, el motor de físicas, la gestión del audio, el sistema de entrada de controles y las herramientas de edición de escenas. El uso de un motor de juego permite a los desarrolladores centrarse en la lógica y el diseño del juego sin tener que programar estos sistemas desde cero.

Indie Game: Es un videojuego desarrollado por un equipo pequeño o de forma independiente, sin el respaldo financiero ni la infraestructura de una gran empresa distribuidora. Los juegos indie se caracterizan por una mayor libertad creativa, presupuestos reducidos y el uso de herramientas accesibles como Godot Engine o Unity. Títulos como Celeste o Super Meat Boy son ejemplos de referencia dentro de este sector.

Speedrun: Es una modalidad de juego que consiste en completar un videojuego o una sección del mismo en el menor tiempo posible. Los jugadores que practican speedrun, conocidos como speedrunners, estudian en profundidad las mecánicas del juego para encontrar rutas óptimas y técnicas que les permitan avanzar más rápido. Un diseño de niveles con una estructura clara y controles precisos, como el de Ascend Tower, favorece este tipo de juego.

Assets: Son todos los recursos digitales que forman parte de un videojuego, incluyendo imágenes, sprites, texturas, modelos, sonidos, música y fuentes tipográficas. Pueden ser creados por el propio equipo de desarrollo o descargados de plataformas externas como la Asset Library de Godot o Itch.io. La calidad y coherencia estética de los assets tiene un impacto directo en la identidad visual del producto final.

Scrum: Es un marco de trabajo ágil para la gestión y desarrollo de proyectos, especialmente utilizado en el desarrollo de software. Se basa en dividir el trabajo en ciclos cortos llamados sprints, con reuniones periódicas de revisión y adaptación. A diferencia del modelo en cascada o waterfall, Scrum permite pivotar y ajustar el proyecto de forma flexible ante cambios o imprevistos durante el desarrollo.

Commit: En el contexto del control de versiones con Git y GitHub, es una operación que guarda un conjunto de cambios realizados en el código o los archivos del proyecto, creando un punto en el historial al que se puede volver en cualquier momento. Cada commit incluye un mensaje descriptivo que explica qué cambios se han realizado, lo que facilita el seguimiento del progreso y la colaboración entre miembros del equipo.

Repositorio: Es el espacio de almacenamiento centralizado donde se guarda el código fuente y todos los archivos de un proyecto de software, junto con el historial completo de cambios realizados. En el contexto de GitHub, un repositorio puede ser público o privado, y permite que varios miembros de un equipo trabajen de forma coordinada sobre el mismo proyecto sin riesgo de pérdida de información o conflictos entre versiones.

Game Feel: Término que hace referencia a la experiencia táctil y sensorial del jugador al interactuar con el videojuego. No se refiere a las reglas o la historia, sino a la "sensación" de los controles: la velocidad de respuesta, la fluidez del movimiento, la inercia del personaje y la retroalimentación visual (partículas, sacudidas de cámara). Un buen *game feel* es lo que hace que el simple hecho de moverse o saltar resulte satisfactorio para el usuario.

6. Bibliografia

[1] *Godot tutorial básico*. YouTube. [Consultado: 2025-2026]

Enlace

[2] *Primeros pasos de Godot*. YouTube. [Consultado: 2025]

Enlace

[3] *Tutorial para hacer los menús en Godot*. YouTube. [Consultado: 2025]

Enlace

[4] *Conceptos básicos del personaje en Godot*. YouTube. [Consultado: 2025]

Enlace

[5] *Plataforma volante en Godot*. YouTube. [Consultado: 2025]

Enlace

[6] *Introducción a los shaders de Godot*. YouTube. [Consultado: 2025]

Enlace

[7] *Explicación de todos los nodos de Godot*. YouTube. [Consultado: 2025]

Enlace

[8] *Movimiento del personaje en Godot*. YouTube. [Consultado: 2025]

Enlace

[9] *Movimiento del personaje en Godot (extenso)*. YouTube. [Consultado: 2025]

Enlace

[10] *Incluir sierras al proyecto en Godot*. YouTube. [Consultado: 2025]

Enlace

[11] *Inspiración para el diseño de niveles*. YouTube. [Consultado: 2025]

Enlace

[12] A7meD. *VHS and CRT monitor effect for Godot 4*. Godot Asset Library. [Consultado: 2025]

Enlace

[13] Godot Engine. *Documentación oficial de Godot 4*. [Consultado: 2025]

Enlace

[14] Suno AI. *Plataforma de generación musical con inteligencia artificial*. [Consultado: 2025]

Enlace

[15] *Asset pack de texturas para plataformas*. Itch.io. [Consultado: 2025]

Enlace

[16] GitHub, Inc. *Plataforma de control de versiones y repositorios*. [Consultado:2025]

Enlace

[17] Godot Engine Foundation. *Godot Engine 4 — Motor de videojuegos de código abierto*. [Consultado:2025]

Enlace

[18] Trello Inc. *Herramienta de gestión de tareas Kanban*. [Consultado: 2024-2025]

Enlace

[19] Itch.io. *Plataforma de distribución de videojuegos independientes*. [Consultado: 2025]

Enlace

[20] Filmora. *Software de edición de vídeo*. Wondershare. [Consultado: 2025]

Enlace

[21] OBS Project. *OBS Studio — Software de grabación y streaming*. [Consultado: 2025]

Enlace

[22] Bongo Cat MR. *Aplicación de visualización de controles en tiempo real*. [Consultado: 2025]

Enlace

[23] Google LLC. *Google Drive — Herramienta de almacenamiento y edición colaborativa en la nube*. [Consultado: 2024-2025]

Enlace

7 Anexos

A continuación se detallan las constantes, variables y funciones principales que controlan el comportamiento del personaje jugable. Cada parámetro ha sido ajustado iterativamente durante las fases de prueba para conseguir un movimiento preciso y fluido, comparable al de los referentes del género.

Script del personaje jugable

```
extends CharacterBody2D
# --- Constantes de Movimiento ---
const SPEED = 256
const ACCELERATION = 2048
const FALL_SPEED = 512
const GRAVITY = 512
# --- Constantes de Salto ---
const JUMP_POWER_INITIAL = -300
const JUMP_POWER = 128
const JUMP_DISTANCE = -256
var jump_time_max = 0.1
var jump_timer = 0
const COYOTE_TIME = 0.1
# --- Constantes de Pared ---
const WALL_JUMP_PUSH = 350
const WALL_SLIDE_START_SPEED = 100
const WALL_SLIDE_ACCEL = 100
var current_wall_friction = 0.0
# --- Variables de Estado ---
var tateobesu = false
var coyote_timer = 0.0
var last_wall: Vector2 = Vector2.ZERO
var _saltando_desde_suelo: bool = false
@export var puede_trepar_paredes: bool = true
```

Figura 50. Script player 1

Constantes de movimiento

SPEED = 256:

Controla la velocidad horizontal del personaje

GRAVITY = 512:

Determina la fuerza con la que el personaje cae

JUMP_POWER_INITIAL = - 300:

Fuerza con la que el personaje sale disparado al saltar

COYOTE_TIME = 0.1:

Permite saltar durante 0.1 segundos aunque ya hayas salido del borde de una plataforma

WALL_JUMP_PUSH = 300:

Fuerza con la que el personaje se aleja de la pared al saltar desde ella

Funciones Principales

```
# --- Muerte ---
signal personaje_muerto
var _muerto: bool
@export var animacion: AnimatedSprite2D
@export var area_2d: Area2D
@export var material_personaje_rojo: ShaderMaterial
func _ready():
	add_to_group("personajes")
	area_2d.body_entered.connect(_on_area_2d_body_entered)
	if ControladorGlobal.tiene_checkpoint:
		global_position = ControladorGlobal.checkpoint
func _on_area_2d_body_entered(_body: Node2D) -> void:
	ControladorGlobal.sumar_muerte()
	animacion.material = material_personaje_rojo
	_muerto = true
	animacion.stop()
	var timer: Timer = Timer.new()
	add_child(timer)
	timer.start(0.2)
	await timer.timeout
	personaje_muerto.emit()
func _physics_process(delta: float) -> void:
	if _muerto:
		return
```

Figura 51. Script player 2

_on_area_2d_body entered:

Se activa al tocar una trampa, cambia el material del personaje a rojo, para la animación y emite la señal de muerte tras 0.2 segundos de delay para que se vea el efecto visual

_muerto = true:

Bloquea cualquier input del jugador una vez muerto para que no pueda seguir moviéndose

_physics_process:

Bucle principal que se ejecuta cada frame, si el personaje está muerto lo ignora todo, si no gestiona gravedad, movimiento horizontal y animaciones en orden

```
	walls(delta)
	_jump(delta)
	# --- GRAVEDAD ---
	if !is_on_floor():
		velocity.y = move_toward(velocity.y, FALL_SPEED, GRAVITY * delta)
		coyote_timer -= delta
	else:
		coyote_timer = COYOTE_TIME
	# --- MOVIMIENTO HORIZONTAL ---
	var direction = Input.get_action_strength("derecha") - Input.get_action_strength("izquierda")
	if direction:
		velocity.x = move_toward(velocity.x, direction * SPEED, ACCELERATION * delta)
	else:
		velocity.x = move_toward(velocity.x, 0, ACCELERATION * delta)
	move_and_slide()
	# --- ANIMACIONES ---
	update_animation_and_direction()
nc _jump(delta: float) -> void:
	if Input.is_action_just_pressed("saltar"):
		tateobesu = true
	if Input.is_action_just_released("saltar"):
		tateobesu = false
	if is_on_floor() or coyote_timer > 0:
		if tateobesu:
```

Figura 52. Script player 3

Sistema de salto:

is_action_just_pressed("saltar"): Detecta el momento exacto en que se pulsa el botón de salto

is_action_just_released("saltar"): Al soltar el botón corta el salto, permitiendo saltos cortos o largos según cuánto se mantenga pulsado

jump_timer: Controla la duración máxima del salto sostenido; cuando llega a 0 deja de aplicar fuerza hacia arriba

JUMP_POWER_INITIAL = -300: aplica la fuerza inicial del salto

JUMP_DISTANCE = -256: fuerza adicional que se aplica mientras se mantiene pulsado el botón

```
        _saltando_desde_suelo = true
        jump_timer = jump_time_max
        var velocidad_plataforma = 0.0
        for i in get_slide_collision_count():
            var colision = get_slide_collision(i)
            if colision.get_collider() is CharacterBody2D:
                velocidad_plataforma = colision.get_collider().velocity.y
        velocity.y = JUMP_POWER_INITIAL - velocidad_plataforma
        tateobesu = false
        coyote_timer = 0
elif is_on_wall() and abs(get_wall_normal().x) > 0.9 and !is_on_floor() and puede_trepar_paredes:
    if tateobesu:
        _saltando_desde_suelo = false
        jump_timer = jump_time_max
        velocity.y = JUMP_POWER_INITIAL
        var wall_normal = get_wall_normal()
        velocity.x = wall_normal.x * WALL_JUMP_PUSH
        last_wall = wall_normal
        tateobesu = false
if Input.is_action_pressed("saltar") and jump_timer > 0:
    velocity.y = move_toward(velocity.y, JUMP_DISTANCE, JUMP_POWER * delta)
    jump_timer -= delta
else:
    jump_timer = -1
```

Figura 53. Script player 4

Sistema de deslizamiento por pared:

current_wall_friction: Va aumentando progresivamente mientras el personaje está pegado a la pared, haciendo que la caída sea cada vez más lenta

WALL_SLIDE_START_SPEED = 32: Velocidad inicial al empezar a deslizarse por la pared

flip_h: Voltea el sprite horizontalmente según la dirección en la que se mueve el personaje

animacion.play(): Selecciona la animación correcta comprobando en orden si está en pared, en el aire, corriendo o quieto

Las sierras son uno de los obstáculos principales del juego. A continuación se describe el comportamiento de su script, que les permite patrullar el entorno de forma autónoma y reaccionar al contacto con el jugador.

```
extends CharacterBody2D

@export var velocidad: float = 200.0
@export var distancia: float = 200.0

var pos_inicial: Vector2
var gravity = ProjectSettings.get_setting("physics/2d/default_gravity")
@onready var sprite = $AnimatedSprite2D

func _ready():
	pos_inicial = global_position
	velocity.x = velocidad
	$raycast_floor_detection.position.x = 20
	$raycast_wall_detection.target_position.x = 20
	sprite.play("default")
	$Area2D.body_entered.connect(_on_area_body_entered)

func _physics_process(delta):
	if not is_on_floor():
		velocity.y += gravity * delta

	if not $raycast_floor_detection.is_colliding():
		velocity.x *= -1
		$raycast_floor_detection.position.x *= -1
		$raycast_wall_detection.target_position.x *= -1
	elif abs(global_position.x - pos_inicial.x) >= distancia:
		velocity.x *= -1
		$raycast_floor_detection.position.x *= -1
		$raycast_wall_detection.target_position.x *= -1

	velocity.x = sign(velocity.x) * velocidad
	move_and_slide()
```

Figura 54. Script de sierra

Constantes y variables:

SPEED:

Velocidad de desplazamiento horizontal de la sierra

gravity:

Gravedad aplicada para que la sierra permanezca pegada al suelo

RAY_FLOOR_POSITION_X:

Posición del rayo que detecta si hay suelo delante; si no lo hay, la sierra gira

RAY_WALL_TARGET_POSITION_X

Distancia del rayo que detecta paredes; si detecta una, la sierra invierte su dirección

velocity.x

Dirección y velocidad horizontal actual de la sierra

Funciones principales:

is_on_floor(): comprueba si la sierra está en contacto con el suelo en cada frame

raycast_floor_detection.is_colliding(): detecta si hay suelo delante; si no lo hay, invierte la dirección de movimiento para evitar que la sierra caiga al vacío

move_and_slide(): aplica el movimiento calculado y gestiona las colisiones con el entorno automáticamente

_on_area_body_entered: se activa cuando la sierra entra en contacto con el jugador, desencadenando la muerte del personaje y el reinicio del nivel

Scripts de controladores

Contador de monedas.

El contador de monedas es el sistema que gestiona la condición de victoria de cada nivel. Su funcionamiento se basa en detectar cuántas monedas hay en la escena al cargarla y compararlo con las que el jugador ha recogido, cargando la siguiente escena cuando ambos valores coinciden.

```
class_name ContenedorMonedas
extends Node


var _total_monedas: int
var _monedas_recogidas: int

# Called when the node enters the scene tree for the first time.
func _ready() -> void:
	var monedas := get_children()
	_total_monedas = monedas.size()
	
	
	for moneda in monedas:
		moneda.contenedor_monedas = self

func moneda_recogida():
	_monedas_recogidas += 1
	
	if _monedas_recogidas == _total_monedas:
		get_parent().get_parent().siguiente_nivel()
```

Figura 55. Contador de monedas.

Variables principales:

_total_monedas:

guarda el número total de monedas que hay en el nivel, calculado automáticamente al iniciarse la escena contando los nodos hijos del contenedor

_monedas_recogidas:

contador que va sumando una unidad cada vez que el jugador recoge una moneda, hasta igualar el total

Funciones principales:

get_children(): obtiene todos los nodos hijos de ContenedorMonedas, que son las monedas del nivel, y cuenta cuántas hay para establecer el total

moneda.contenedor_monedas = self: conecta cada moneda con este script para que puedan comunicarse y notificar cuando son recogidas

moneda_recogida(): función que se llama cada vez que el jugador toca una moneda; suma 1 al contador y comprueba si se han recogido todas

if _monedas_recogidas == _total_monedas: comprueba si el contador de monedas recogidas iguala al total del nivel; si es así, llama a siguiente_nivel() para cargar la siguiente escena de la torre

Repositorios de GitHub

Durante el desarrollo se mantuvieron dos repositorios diferenciados en GitHub, cada uno con un propósito específico dentro del proyecto.

Repositorio de la web

Este repositorio aloja la página web pública del proyecto mediante GitHub Pages. Está compuesto por un archivo index.html, una hoja de estilos styles.css y los recursos visuales de la página. Su función es servir como punto de acceso para cualquier usuario externo: desde aquí se puede descargar el juego, conocer el proyecto y enviar sugerencias o comentarios directamente al equipo a través del formulario de contacto integrado con Formspree.

Repositorio del proyecto

Este repositorio contiene el código fuente completo del juego desarrollado en Godot Engine. Se utilizó como herramienta de control de versiones durante todo el desarrollo, permitiendo subir avances semanalmente y trabajar de forma coordinada entre los dos miembros del equipo sin riesgo de perder o sobreescribir el trabajo del otro. Incluye todos los scripts en GDScript, las escenas, los assets y los recursos del proyecto.

Envío de opiniones, bugs y sugerencias

Para que los jugadores pudieran comunicarse con el equipo de desarrollo de forma sencilla, se implementó en la página web un formulario de contacto mediante Formspree, un servicio externo que permite recibir mensajes directamente en el correo electrónico sin necesidad de un servidor propio. El usuario únicamente debe introducir su nombre, su correo electrónico y su mensaje, y al pulsar el botón de envío el equipo lo recibe de forma inmediata en su bandeja de entrada.

¿Bugs? ¿Sugerencias?

Adrián López Ruiz

alopezr@elpuig.xeill.net

Prueba para el proyecto

ENVIAR A LA TORRE

Figura 56. Envío de opinión

New form submission on Proyecyo

Someone just submitted a form on adrifroste.github.io/. Here's what they had to say:

name
Adrián López Ruiz

email
alopezr@elpuig.xeill.net

message
Prueba para el proyecto

Figura 57. Recibir opinión

Para implementarlo se registró el correo del equipo en Formspree y se añadió una sección específica en el index.html. A continuación se describen los atributos principales del formulario:

action: indica la URL de Formspree a la que se envían los datos del formulario; incluye el ID de la cuenta registrada

method="POST": los datos se envían de forma oculta, sin que aparezcan en la URL del navegador

type="text" / type="email": definen el tipo de campo; el tipo email valida automáticamente que el valor introducido contenga el símbolo @ antes de permitir el envío

name: etiqueta identificativa con la que llega cada dato al correo del equipo

placeholder: texto gris de ejemplo que aparece dentro del campo cuando está vacío, orientando al usuario sobre qué debe introducir

required: Atributo que impide enviar el formulario si el campo está vacío, asegurando que no lleguen mensajes incompletos

rows: Define la altura visual del área de texto en número de líneas

type="submit": Define el botón que, al pulsarse, envía todos los datos del formulario al servidor de Formspree

```
        <h2 class="sc-section-title reveal">¿Bugs? ¿Sugerencias?</h2>
<div class="sc-form-card reveal" id="contact">
    <form action="https://formspree.io/f/xpqonjya" method="POST" class="dark-form">
        <div class="form-row">
            <input type="text" name="name" placeholder="Nombre / Gremio" required>
            <input type="email" name="email" placeholder="Email" required>
        </div>
        <textarea name="message" rows="5" placeholder="Escribe tu mensaje, hereje..." required></textarea>
        <button type="submit" class="sc-btn sc-btn-cyan" style="width:fit-content;">Enviar a la torre</button>
    </form>
```

Figura 58. Script de envío de mensajes

Bloques trampa

Los bloques trampa son plataformas frágiles que se derrumban al entrar en contacto con el jugador, añadiendo presión y urgencia al movimiento. Su presencia en el nivel obliga al jugador a planificar su ruta con antelación, ya que detenerse sobre uno de ellos garantiza la caída. Esta mecánica fuerza una toma de decisiones rápida y precisa: el jugador debe saber exactamente a dónde saltar antes de que el bloque desaparezca bajo sus pies, convirtiendo cada plataforma frágil en un reto de reacción y memoria que eleva la dificultad de forma natural sin necesidad de añadir más obstáculos.

```
extends StaticBody2D

func _ready():
	$Area2D.body_entered.connect(_on_body_entered)
	$GPUParticles2D.emitting = false

func _on_body_entered(body):
	if body.is_in_group("personajes"):
		_temblar()
		await get_tree().create_timer(0.5).timeout
		$Sprite2D.visible = false
		$CollisionShape2D.set_deferred("disabled", true)
		$GPUParticles2D.emitting = true
		await get_tree().create_timer(0.5).timeout
		queue_free()

func _temblar():
	var pos_original = $Sprite2D.position
	var tween = create_tween()
	tween.set_loops(10)
	tween.tween_property($Sprite2D, "position", pos_original + Vector2(1, 0), 0.05)
	tween.tween_property($Sprite2D, "position", pos_original + Vector2(-1, 0), 0.05)
```

Figura 59. Script de bloques trampa

Funciones principales

_ready(): Conecta la señal de contacto y apaga las partículas.

_on_body_entered(body): Si toca un personaje: tiembla → oculta → desactiva colisión → partículas → se destruye.

_temblar(): Vibra 10 veces (1px izq/der, 0.05s) antes de desaparecer.