Institut Puig Castellar
Santa Coloma de Gramenet

RampTruck

(Proyecto de Desarrollo)
CFGM Sistemas Microinformáticos y Redes

Autores	Rayan El Maazouzi Raho / Abdelkader Hamza Mostefi
Curso	2SMX
Grupo	A
Juego	https://rayan-tech619.github.io/RampTruck-Web/

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 © 2026 Rayan El Maazouzi Raho / Abdelkader Hamza Mostefi Hakoum

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

© (Rayan El Maazouzi Raho / Abdelkader Hamza Mostefi Hakoum)
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

Este juego es una experiencia 2D desarrollada con Godot en la que el jugador toma el control de un coche y debe avanzar a través de una serie de niveles diseñados para poner a prueba su habilidad. El objetivo principal es sencillo en concepto, pero desafiante en la práctica: llegar a la meta sin que el coche se quede atrapado, volcado o atascado en el recorrido.

Cada nivel combina elementos de parkour y precisión, obligando al jugador a dominar aspectos como la aceleración, el frenado, el equilibrio y el control en el aire. No se trata solo de avanzar, sino de hacerlo con técnica y anticipación. Las rampas, plataformas elevadas, pendientes pronunciadas y distintos obstáculos están colocados estratégicamente para exigir reflejos rápidos y una planificación cuidadosa de cada movimiento.

La física juega un papel fundamental en la experiencia, ya que cada salto y cada aterrizaje pueden marcar la diferencia entre continuar avanzando o tener que intentarlo de nuevo. Mantener la estabilidad del coche, calcular la velocidad adecuada antes de un salto y reaccionar ante terrenos irregulares son habilidades clave para superar los desafíos que propone el juego.

A medida que se avanza, los niveles se vuelven progresivamente más complejos y exigentes, con nuevos obstáculos, trayectos más técnicos y secciones que requieren un control aún más preciso. Esto mantiene la experiencia dinámica y motivadora, invitando al jugador a mejorar constantemente su rendimiento y perfeccionar su técnica hasta completar todos los desafíos y alcanzar la meta final.

Palabras clave

Conducción, precisión, habilidad, desafío, equilibrio, control, niveles, rampas.

Abstract

This game is a 2D experience developed with Godot in which the player takes control of a vehicle and must progress through a series of levels designed to test their skill. The main objective is simple in concept but challenging in practice: reach the finish line without the vehicle getting stuck, flipping over, or becoming trapped along the route.

Each level combines elements of parkour and precision, forcing the player to master aspects such as acceleration, braking, balance, and mid-air control. It is not just about moving forward; it is about doing so with technique and anticipation. Ramps, elevated platforms, steep slopes, and various obstacles are strategically placed to demand quick reflexes and careful planning of every movement.

Physics play a fundamental role in the experience, as every jump and landing can make the difference between moving forward or having to try again. Maintaining the vehicle's stability, calculating the appropriate speed before a jump, and reacting to irregular terrain are key skills for overcoming the challenges proposed by the game.

As the player progresses, the levels become increasingly complex and demanding, featuring new obstacles, more technical paths, and sections that require even more precise control. This keeps the experience dynamic and motivating, inviting the player to constantly improve their performance and perfect their technique until they complete all challenges and reach the final goal.

Keywords

Driving, Precision, Skill-based, Parkour, Challenge, Balance, Control, Levels

Índice

Lista de Figuras

1.- Introducción

El presente proyecto consiste en el diseño y desarrollo de Ramp Truck, un videojuego de conducción y precisión en 2D desarrollado íntegramente en el motor Godot Engine 4. A diferencia de los juegos de carreras convencionales, este título se centra en la superación de niveles mediante el control de la física, la gestión del impulso y la destrucción de obstáculos.

El jugador controla un vehículo articulado que debe navegar por terrenos irregulares, evitar trampas mortales (como prensas hidráulicas y plantas carnívoras) y utilizar mecánicas de disparo y propulsión (*boost*) para llegar a la meta. El núcleo del juego reside en la dificultad técnica y la precisión necesaria para mantener la estabilidad del vehículo basado en cuerpos rígidos (*RigidBody2D*).

1.2 Justificación del cambio de concepto

Tras una fase inicial de investigación sobre géneros de terror y mecánicas de *Escape Room*, se decidió pivotar el proyecto hacia un género de conducción física. Este cambio se realizó con el objetivo de profundizar en el uso de motores de física avanzada, sistemas de partículas y la gestión compleja de colisiones, elementos que ofrecían un reto técnico más alineado con los objetivos del ciclo formativo de Sistemas Microinformáticos y Redes.

1.1 Contexto

En la actualidad, la industria del videojuego se encuentra en una etapa de gran expansión y evolución tecnológica. El acceso a motores gráficos gratuitos y de código abierto como Godot Engine ha permitido que pequeños equipos e incluso desarrolladores individuales puedan crear videojuegos completos con un nivel técnico y visual cada vez más profesional.

Dentro del sector, los videojuegos basados en físicas realistas y desafíos de precisión han mantenido una popularidad constante, especialmente en el ámbito de los dispositivos móviles y plataformas de escritorio. Este tipo de experiencias se caracterizan por combinar la gestión del impulso, el control de cuerpos rígidos y la superación de obstáculos mecánicos. A diferencia de otros géneros puramente lineales, los juegos de conducción técnica priorizan la habilidad del jugador y su capacidad para interactuar con un entorno dinámico y destructible.

Por otro lado, el desarrollo de videojuegos se ha convertido en una herramienta educativa de gran valor, ya que integra múltiples disciplinas técnicas: programación en GDScript, diseño de sistemas físicos, lógica computacional, gestión de señales y planificación de proyectos. Además, permite aplicar conocimientos teóricos en un entorno práctico y creativo, donde el desarrollador debe optimizar la estabilidad del juego ante cálculos de física complejos.

En este contexto, la elección de desarrollar un juego de conducción y habilidad como Ramp Truck responde tanto a la vigencia de los retos arcade en el mercado actual como a la oportunidad formativa que ofrece. El proyecto se sitúa dentro del ámbito del desarrollo independiente (indie), donde la innovación en las mecánicas y la correcta implementación de las leyes de la física son factores clave para el éxito del título.

La combinación de accesibilidad tecnológica, el interés por las mecánicas de juego basadas en la superación de retos y la posibilidad de aplicar competencias técnicas adquiridas durante el ciclo formativo justifican y motivan la ejecución de este proyecto.

1.2 Justificación

El desarrollo de este proyecto es relevante tanto desde el punto de vista académico como técnico y creativo.

En primer lugar, a nivel académico, permite aplicar de manera práctica los conocimientos adquiridos durante el ciclo formativo de Sistemas Microinformáticos y Redes. El proyecto integra competencias como la programación en GDScript, la planificación de tareas, la gestión de recursos, el control de versiones y la organización del trabajo en equipo. Esto supone una oportunidad para consolidar los aprendizajes teóricos mediante una aplicación real y funcional en un entorno de desarrollo profesional.

En segundo lugar, desde un punto de vista técnico, el desarrollo de Ramp Truck conlleva un nivel de complejidad elevado debido al uso de sistemas de física avanzada. Implica trabajar con el motor Godot Engine, gestionar el comportamiento de cuerpos rígidos (RigidBody2D), implementar articulaciones mecánicas (PinJoint2D), controlar colisiones complejas, desarrollar sistemas de partículas para las explosiones y optimizar el rendimiento del programa para garantizar una conducción fluida. Todo esto requiere una gran capacidad de análisis, resolución de problemas y una estructuración correcta del código.

Además, el género escogido —un juego de conducción y precisión en 2D— es una propuesta muy consolidada y demandada dentro del sector de los videojuegos independientes y para dispositivos móviles. Este tipo de experiencias ofrecen un reto constante al jugador mediante la superación de obstáculos y la gestión de la habilidad bajo presión. Esto convierte al proyecto en una propuesta atractiva, divertida y alineada con las tendencias actuales del mercado de los juegos de tipo arcade y puzles físicos.

Finalmente, el proyecto también tiene un valor creativo importante, ya que permite desarrollar aspectos como el diseño de niveles (Level Design), la creación de mecánicas de destrucción y la planificación de una experiencia interactiva coherente y satisfactoria. Esta combinación de técnica, física y creatividad aporta un valor añadido al trabajo y lo convierte en una experiencia formativa completa.

Por todos estos motivos, se considera que el proyecto es pertinente, viable y adecuado para demostrar las competencias adquiridas durante el ciclo formativo.

1.3 Objetivos

1.3.1 Objetivo general

El objetivo principal es diseñar y desarrollar un videojuego de conducción técnica en 2D que ponga a prueba la destreza manual, el cálculo de trayectorias físicas y la capacidad de reacción del jugador.

Se busca crear una experiencia interactiva donde el dominio de las inercias y la resolución de obstáculos mecánicos superen la jugabilidad de los títulos de conducción arcade convencionales.

1.3.2 Objetivos específicos

- **Implementar un sistema de físicas avanzado**: Configurar el comportamiento de un vehículo basado en cuerpos rígidos (RigidBody2D) para lograr una conducción realista y desafiante.
- **Diseñar niveles de precisión**: Crear escenarios con rampas, terrenos irregulares y trampas mecánicas que obliguen al jugador a gestionar la aceleración y el equilibrio bajo presión.
- **Desarrollar mecánicas de interacción dinámica**: Programar obstáculos reactivos, como prensas hidráulicas y plantas carnívoras, que interactúen directamente con el estado del vehículo.
- **Optimizar el feedback visual y de cámara**: Implementar un sistema de cámara cinematográfica con zooms dinámicos y efectos de fragmentación procedimental para mejorar la inmersión tras una derrota.
- **Integrar sistemas de combate y propulsión**: Desarrollar lógicas de disparo de proyectiles y un sistema de Boost por Doble Tap para añadir una capa estratégica a la movilidad del vehículo.

1.4 Estrategia y planificación del proyecto

Para la ejecución de este proyecto, se han considerado diversas vías de actuación, desde la modificación de proyectos de código abierto hasta la creación de un sistema propio. A continuación, se detalla la estrategia elegida y su viabilidad.

1.4.1 Análisis de estrategias posibles

Se valoraron las siguientes opciones estratégicas:

- **Adaptación de un producto existente**: Utilizar una plantilla de conducción 2D ya creada en Godot. Aunque reducía los tiempos de desarrollo, limitaba nuestra capacidad para implementar físicas personalizadas de cuerpos rígidos y mecánicas de interacción avanzada con el entorno.
- **Desarrollo de un producto nuevo (Estrategia elegida)**: Crear el videojuego desde una escena vacía, diseñando desde cero la jerarquía de nodos, los scripts de movimiento y el diseño de niveles. Esta es la opción más ambiciosa y la que permite una personalización total de la experiencia técnica y visual.

1.4.2 Justificación de la estrategia (Estudio de viabilidad)

Hemos optado por el desarrollo de un producto nuevo inspirado en dos grandes referentes: los clásicos juegos de acrobacias de motos (como *Moto X3M*) y los títulos de plataformas de precisión física (como *Hill Climb Racing*). La elección de esta estrategia se basa en los siguientes puntos de viabilidad:

- **Diferenciación estética y mecánica**: Al desarrollar un producto propio, podemos fusionar la jugabilidad de superación de obstáculos con una mecánica de disparo de proyectiles y un control de inercia detallado, logrando una propuesta original dentro del género de conducción 2D.
- **Independencia de red (Offline)**: El análisis de mercado indica una alta demanda de juegos "sin conexión". Nuestra estrategia se centra en un producto que no dependa de servidores, garantizando la jugabilidad en cualquier entorno.
- **Viabilidad técnica en Godot 4**: El uso de nodos como CollisionPolygon y el sistema de señales de Godot permite gestionar un desarrollo profesional con recursos optimizados, haciendo que el proyecto sea técnicamente viable y escalable.
- **Control de la progresión**: Crear el sistema desde cero permite ajustar con precisión la curva de dificultad mediante el ensayo y error, asegurando que el reto de equilibrar el vehículo y destruir trampas sea adictivo y fluido.

Esta estrategia es la más apropiada porque garantiza que cada rampa, cada diamante y cada trampa mecánica estén integrados en un código limpio, optimizado y totalmente original.

1.5 Metodología de trabajo

Para asegurar el éxito del desarrollo y el cumplimiento de los plazos establecidos, se ha optado por seguir una metodología tradicional en cascada (estilo Waterfall). Esta elección permite mantener un control estricto sobre cada etapa del ciclo de vida del videojuego, asegurando que los cimientos técnicos estén sólidos antes de avanzar a la creación de contenido.

1.5.1 Argumentación de la metodología

Se considera que esta es la metodología más apropiada por las siguientes razones:

- **Estructura lineal y clara:** Al ser un equipo de desarrollo pequeño, tener fases bien delimitadas (Análisis, Diseño, Implementación, Testing y Lanzamiento) evita la duplicidad de tareas y permite centrar los esfuerzos en un solo objetivo a la vez.

- **Dependencias técnicas:** En un videojuego, muchas tareas dependen de la anterior; por ejemplo, no se pueden diseñar los niveles (Mundo 1 y 2) sin haber programado previamente el script de movimiento y las colisiones del vehículo.

- **Previsibilidad*:*** Permite identificar desde el inicio los hitos clave del proyecto, facilitando el control sobre el progreso global.

1.5.2 Herramientas de seguimiento

La herramienta principal para la gestión y el seguimiento del proyecto es el Diagrama de Gantt. Esta herramienta es fundamental en nuestra metodología por los siguientes motivos:

- **Planificación temporal:** Nos permite visualizar de forma gráfica la duración de cada tarea y cómo se distribuyen a lo largo del calendario escolar.

- **Gestión de dependencias:** A través del diagrama, podemos observar qué tareas deben finalizar para que otras puedan comenzar, evitando cuellos de botella en la programación de Godot.

- **Control de hitos:** Facilita la identificación de fechas críticas, como la entrega de la versión alfa (mecánicas básicas) o la versión beta (niveles completos).

Además del diagrama de Gantt, se utilizan un control de versiones para asegurar que cada actualización esté debidamente registrada y validada antes de pasar a la siguiente.

1.6 Estudio económico y presupuestario

Para poder llevar a cabo el desarrollo de RampTruck, hemos evaluado y contabilizado todos los recursos materiales, herramientas informáticas y periféricos que hemos utilizado a lo largo del curso. A continuación, se detalla el presupuesto estimado del proyecto, diferenciando aquellos recursos de uso personal, los proporcionados por el centro de estudios y el software libre empleado:

Producto	Descripción	Precio Final
Ordenador Gaming	Usado por nosotros para desarrollar y testear el proyecto desde casa con un rendimiento óptimo.	1000€ aprox.
Ordenador de clase	Equipos informáticos proporcionados por el instituto que hemos usado para programar y trabajar en grupo en el aula.	500€ aprox.
Godot Engine 4	Motor gráfico principal de código abierto que hemos elegido para desarrollar todo el videojuego en 2D.	0€
Teclado	Periférico de entrada esencial que hemos utilizado tanto para escribir el código fuente como para las pruebas de control.	25€ aprox.
Ratón	Periférico de entrada utilizado de forma constante para movernos por la interfaz de Godot y diseñar los niveles.	10€ aprox.
GitHub	Plataforma de almacenamiento y control de versiones en la nube que nos ha permitido organizar el código de forma segura.	0€

Recursos gráficos y sonoros	Conjunto de sprites (texturas de hielo, tierra, coche) y efectos de sonido que hemos obtenido de forma gratuita.	0€
YouTube y documentación	Plataforma audiovisual y guías en línea utilizadas para consultar tutoriales, buscar información técnica e inspirarnos para los scripts.	0€

1.6.1 Conclusión del estudio económico

Aunque el valor total de los materiales y herramientas utilizados asciende teóricamente a 1.735€ aprox., el coste real y directo para el desarrollo de este proyecto ha sido de 0€. Esto se debe a que todos los materiales físicos (como los ordenadores de casa, el portátil y los periféricos) ya eran de nuestra propiedad con anterioridad o bien han sido facilitados por el propio centro educativo para trabajar en clase. Por lo tanto, no hemos tenido que realizar ninguna inversión económica nueva para llevar a cabo el videojuego, aprovechando además que tanto el motor gráfico Godot, la plataforma GitHub, los tutoriales de consulta y los recursos artísticos son completamente gratuitos y de libre acceso.

2 Descripción del proyecto

2.1.1 Requisitos funcionales

El desarrollo del videojuego se basa en una serie de pilares fundamentales que garantizan que la experiencia sea interactiva y desafiante. No se trata solo de mover un objeto, sino de crear un sistema complejo de reglas:

- **Control Físico del Vehículo:** El requisito principal es que el jugador sienta el control sobre una máquina. Esto implica que el software debe interpretar las pulsaciones de teclado para aplicar fuerzas de aceleración. Además, el control no termina cuando el coche deja de tocar el suelo; hemos implementado una mecánica de equilibrio aéreo que permite al usuario rotar el chasis. Esto es vital para la supervivencia, ya que el juego debe detectar si el aterrizaje se hace sobre las ruedas (continuar) o sobre el techo (derrota).

- **Sistema Dinámico de Coleccionables:** El juego debe gestionar una base de datos interna de los objetos recogidos. Al entrar en contacto con un diamante, el sistema no solo debe sumar un punto, sino que debe gestionar una serie de eventos visuales y sonoros. El requisito es que el diamante actúe como un sensor inteligente que se comunica con el marcador global del jugador para actualizar el progreso en tiempo real.

- **Flujo de Navegación entre Escenas:** Un requisito clave es la capacidad de cambiar entre diferentes interfaces del juego sin errores. Esto incluye la carga de menús, la pantalla de selección de mundos y la gestión de las pantallas de "Game Over" o "Victoria". El sistema debe ser capaz de limpiar la memoria de la escena anterior y cargar los nuevos recursos de forma eficiente para evitar tirones en el rendimiento.

- **Sistema de Cámara con Restricciones:** Para evitar que el jugador se desoriente, la cámara debe tener una lógica de seguimiento suave. Un requisito específico es la configuración de límites técnicos; la cámara se detendrá automáticamente al llegar a los bordes diseñados del nivel, ocultando así el "vacío" del motor de juego y manteniendo la inmersión en todo momento.

2.1.2 Requisitos no funcionales

- **Juego 3D:** Al empezar el proyecto habíamos empezado haciendo un juego 3d, pero a medida que avanzábamos el personaje principal no se movía, a lo mejor era tema colisiones, pero al final como no entendíamos y los profesores dijeron que era mejor un juego 2D nos decantamos a eso.

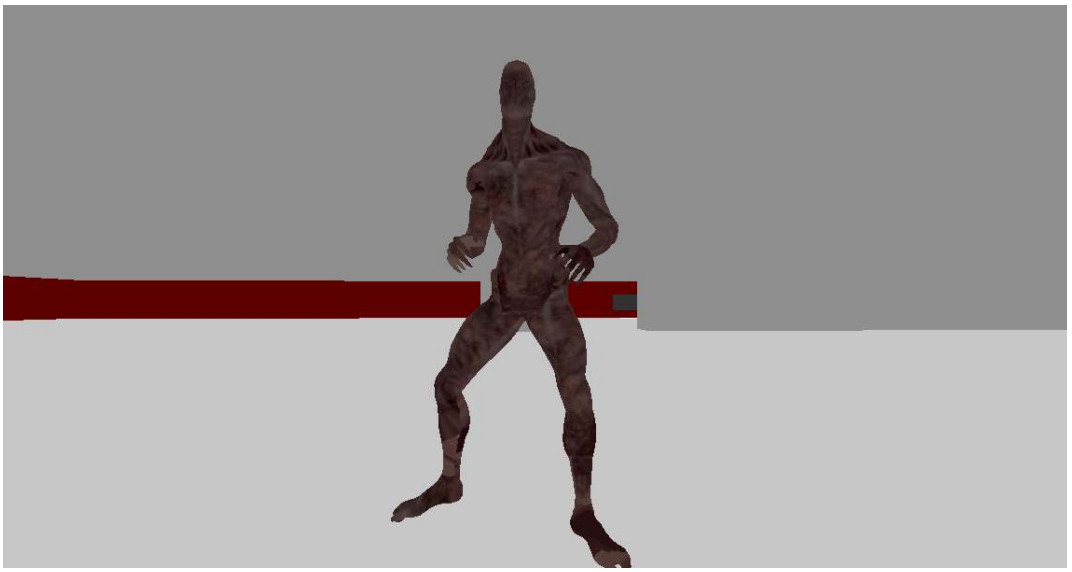


Foto del Monstruo del anterior juego

- **Evolución de Mecánicas (Sustitución de Salto)**: El requisito original de implementar un salto vertical mediante la tecla espacio fue descartado tras detectar inconsistencias en el motor de físicas al combinarlo con la rotación aérea de **Rigidbody2D.** En su lugar, se optó por potenciar el **Boost por Doble Tap** y añadir una mecánica de **Disparo de Proyectiles**. Esta decisión estratégica ofrece un control más dinámico y compensa la movilidad vertical sin comprometer la estabilidad del chasis.

- **Pantalla de victoria animada:** Queríamos implementar una pantalla de victoria al pasar un nivel con una buena animación, pero no sabíamos como hacerlo ni en que basarnos para que el jugador reciba la puntuación.

- **Rampa de bucle** : Deseábamos añadir como una estructura que sea una rampa de bucle que el jugador tenga que dar una vuelta y seguir en línea recta, pero debido a su complejidad no supimos hacerla.

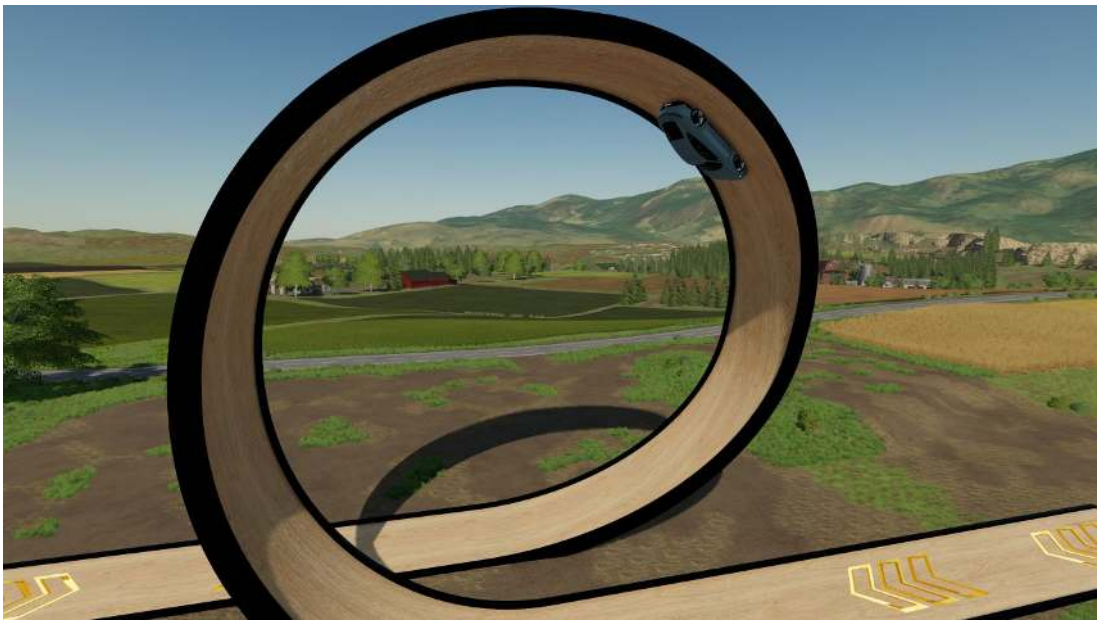


Foto de la Rampa de Bucle

2.2 Tecnologías

2.2.1 Comparativa de las tecnologías valoradas

Durante la fase de investigación, nos planteamos seriamente qué herramientas utilizar. Analizamos Roblox Studio por su gran capacidad para compartir juegos rápidamente, pero nos dimos cuenta de que su sistema de físicas está muy orientado al 3D y nos daba muchos problemas para conseguir la precisión técnica que buscábamos en un entorno 2D. También valoramos otros motores como Unity, pero finalmente nos decidimos por Godot Engine 4. La razón principal es que los profesores nos lo aconsejaron, pero también Godot es extremadamente ligero y su sistema de nodos es ideal para proyectos donde quieres tener un control absoluto sobre cada parte del código y del arte visual sin complicaciones innecesarias.

2.2.2 Tecnologías escogidas

- **Godot 4**: Es la herramienta sobre la que se asienta todo el proyecto. Su gran ventaja es que permite separar el juego en "Escenas". Por ejemplo, el coche es una escena que podemos probar por separado y luego meterla en cualquier nivel. Esto nos ha ahorrado mucho tiempo de desarrollo.

- **GDScript**: Es el lenguaje de programación que hemos usado. Al ser tan parecido a Python, nos ha permitido escribir lógicas complejas de forma muy legible. Gracias a él, pudimos programar el "cerebro" de los enemigos y el comportamiento de las rampas de forma ágil.

2.4 Descripción de los componentes

2.4.1 El Jugador: Arquitectura del Coche

El coche no es un bloque sólido, sino una pieza de ingeniería virtual. Está compuesto por una Carrocería principal y dos nodos que son las Ruedas. Lo más interesante es cómo hemos programado la suspensión: mediante código, las ruedas detectan la distancia al suelo y aplican una fuerza hacia arriba al chasis. Esto permite que el coche rebote y se incline de forma realista al subir rampas o caer de grandes saltos, dando una sensación de peso y potencia al motor del vehículo.

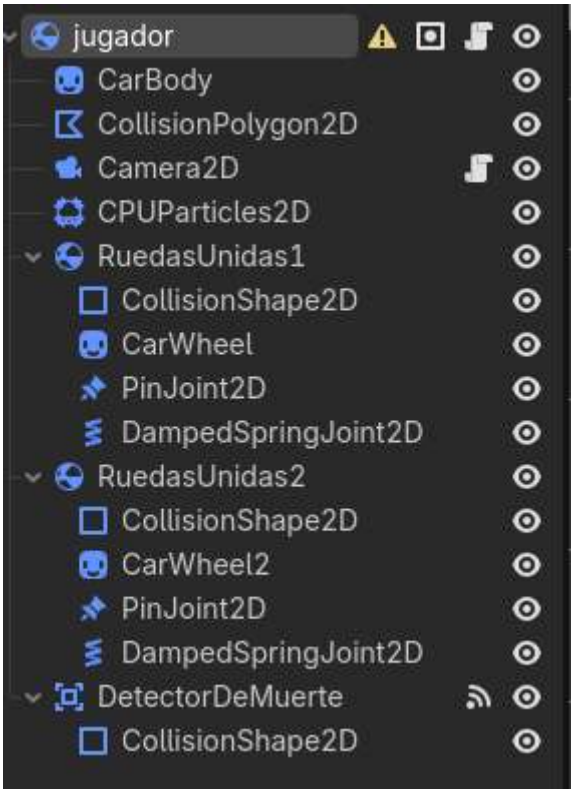


Foto de la estructura

También para conectarlas, se arrastró y posicionó un PinJoint2D justo en el centro de cada rueda

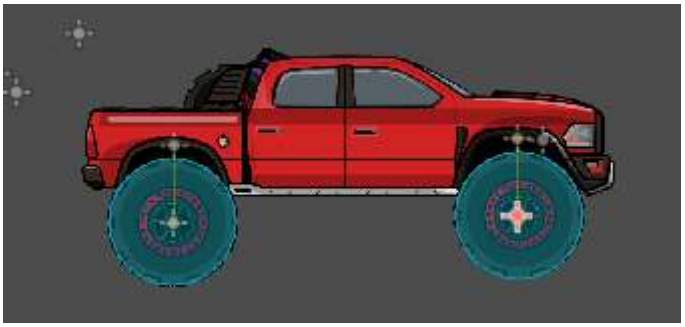


Foto de la carrocería con sus PinJoint2D

Además, se añadió un DampedSpringJoint2D (muelle) desplazado hacia arriba para que la carrocería pueda bajar y subir sin separarse de las ruedas, creando el efecto de amortiguación.

El efecto de amortiguación más concretamente es desde el medio de la rueda hasta la parte que va arriba pegada a la carrocería ese es el trayecto máximo que podrían hacer las ruedas al rebotar.

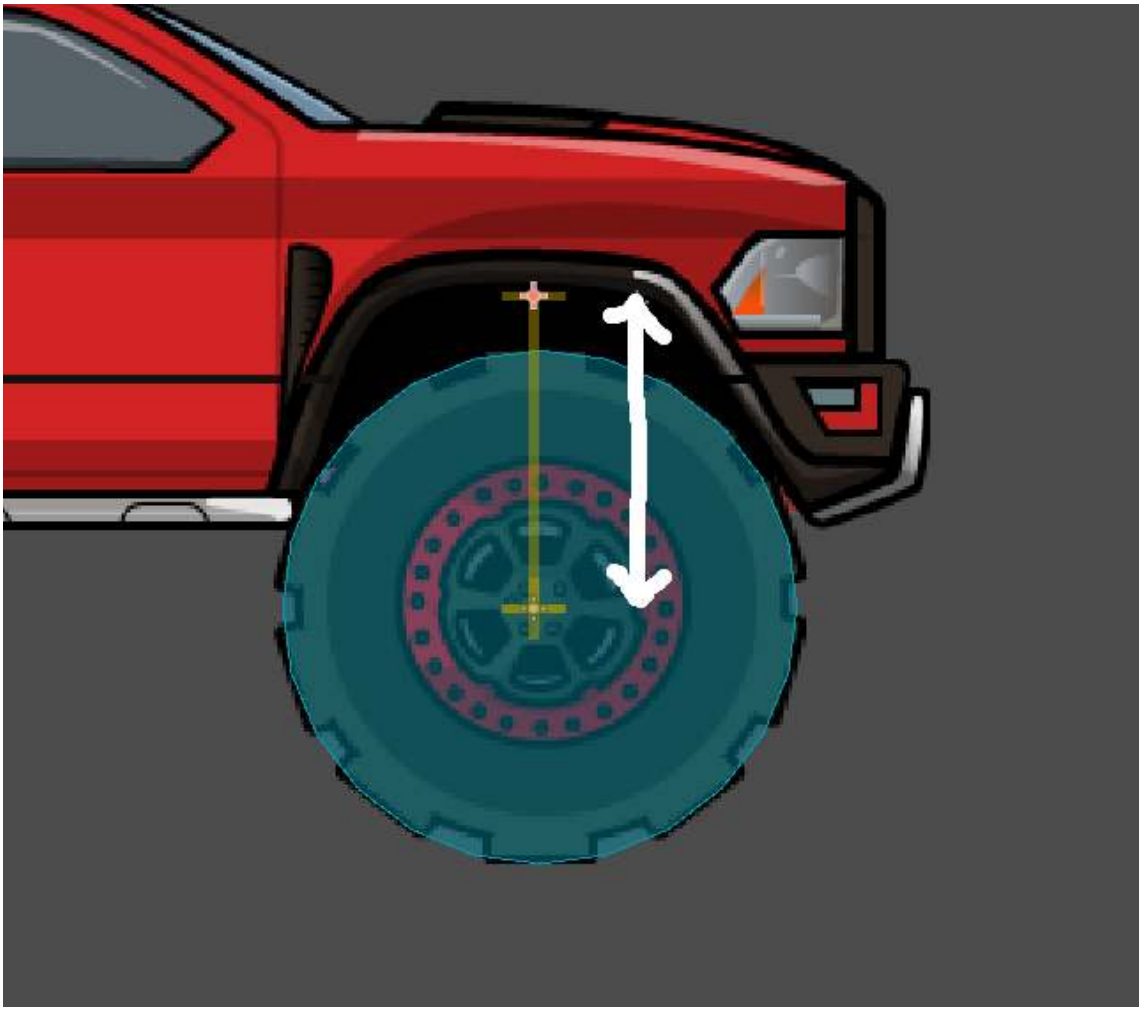


Foto mostrando el rango de amortiguación

2.4.2 Componente de Entorno: Construcción del Nivel y Colisiones (Floor)

El diseño y la implementación del circuito no se han realizado como una pieza única, sino que se han dividido en dos capas de trabajo totalmente diferenciadas: la capa visual (lo que el jugador ve) y la capa física (donde el coche interactúa). Esta separación es fundamental para lograr un equilibrio entre un apartado artístico atractivo y una jugabilidad técnica precisa.

A. Diseño y Estética Visual

Para la construcción del escenario que el jugador recorre, se ha utilizado el nodo TileMapLayer. Esta herramienta permite "pintar" el mapa utilizando un TileSet específico, que en nuestro caso está compuesto por piezas temáticas de hielo y tierra. El diseño se apoyó en una rejilla técnica que garantiza la alineación perfecta de todas las plataformas. Esto evita que existan huecos visuales entre los bloques de tierra y asegura que el mundo tenga una coherencia estructural profesional. Al usar baldosas (tiles) de hielo y tierra, el motor gráfico de Godot solo tiene que cargar unas pocas imágenes pequeñas que se repiten, lo que permite crear niveles muy largos sin que el ordenador sufra por falta de memoria. También se ha implementado el **Parallax Background**, compuesto por capas que se mueven a distintas velocidades para crear profundidad.

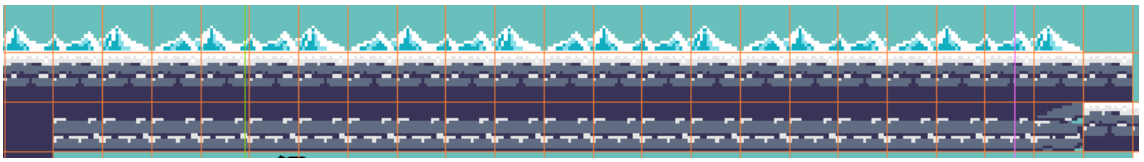


Foto de las rejillas del TileMapLayer

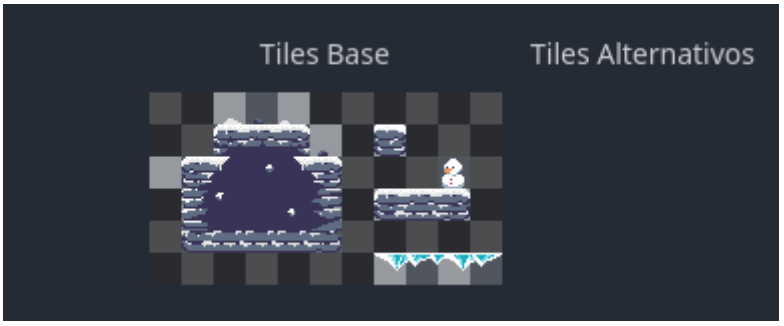


Foto mostrando el sprite en el TileBase

B. Capa de Colisiones Manuales y Precisión

Uno de los mayores retos del proyecto fue conseguir que el coche se deslizara suavemente. Las colisiones automáticas que vienen por defecto en los TileMaps suelen ser cuadradas y toscas, lo que provocaba que el coche diera saltos bruscos o se quedara atascado en las uniones de los bloques. Por ello, decidimos ignorar las colisiones automáticas y crear nodos manuales de alta precisión mediante el **CollisionPolygon2D**. En lugar de usar cajas, trazamos polígonos a medida que siguen la forma exacta del dibujo. Esto es lo que permite que existan rampas suaves; el coche puede subir pendientes inclinadas de forma fluida, manteniendo siempre el contacto con el suelo de manera realista.

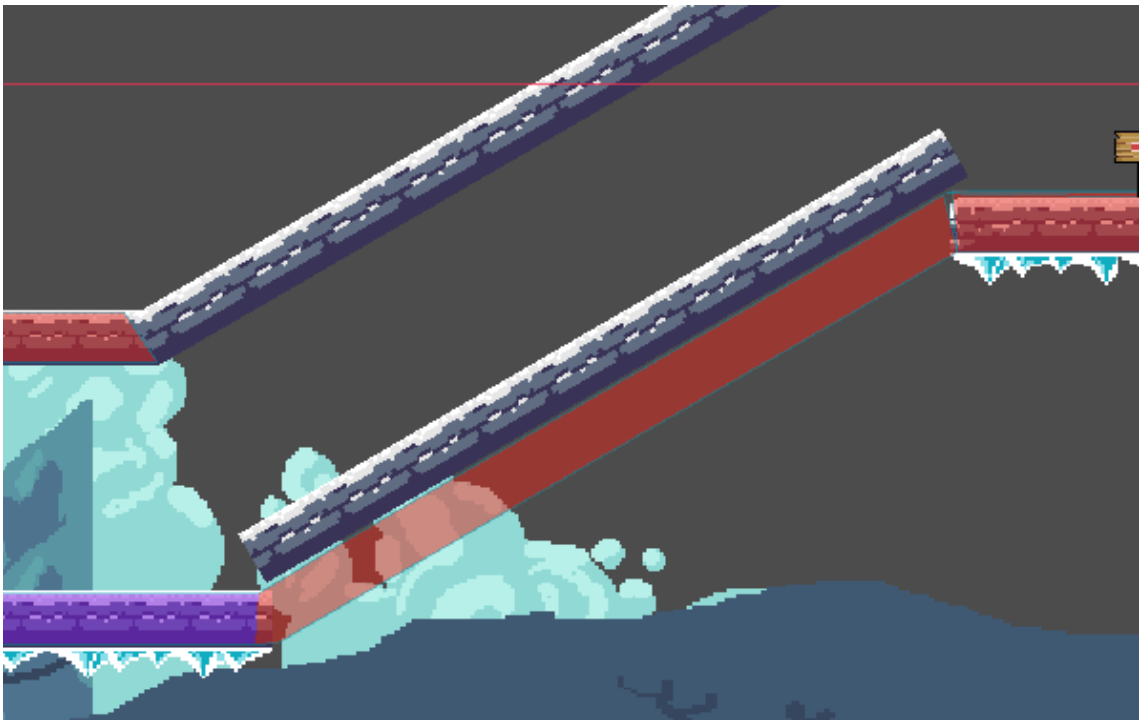


Foto mostrando la rampa en diagonal y su colisión

Una técnica específica que aplicamos fue la extracción de secciones del suelo del TileMapLayer. Tomamos una foto o captura de la textura del suelo, la rotamos hacia un lado para darle la inclinación deseada y, posteriormente, le añadimos su colisión manual correspondiente. Esto nos permitió crear rampas con ángulos de ataque perfectos para el salto del vehículo.

B. Nodos de Túnel (Estructura Doble):

Para los niveles de mayor dificultad, como el Mundo 1, diseñamos nodos de túnel diferenciados. Separamos técnicamente el Tunnel techo del Tunnel suelo. Esta distinción es vital porque permite que el coche pueda rodar por espacios extremadamente estrechos manteniendo una detección de colisión exacta en ambos lados simultáneamente.

C. Creación básica dentro del world:

La organización del proyecto es jerárquica. La escena principal, que llamamos "World", es el contenedor de todo lo que el jugador ve. Para que el proyecto no fuera un caos de objetos, lo dividimos en capas lógicas:

- **El Fondo (Efecto Parallax):** Explicado detalladamente, consiste en 4 capas diferentes (cielo, nubes, montañas lejanas y siluetas cercanas). Cada capa se mueve a una velocidad distinta respecto al coche. Este detalle técnico crea una ilusión de profundidad muy potente; parece que el mundo es profundo y realista en lugar de un simple dibujo plano.

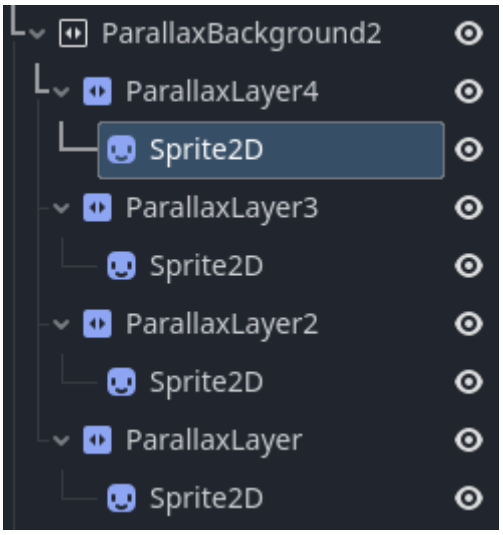


Foto de la estructura jerárquica de las capas

Foto de la imagen del Fondo Parallax

- **Capa de Gameplay**: Aquí es donde vive el coche y el terreno. Hemos separado los elementos interactuables (como diamantes y metas) de los elementos estáticos (como el suelo) para que el motor de físicas de Godot pueda procesar las colisiones de forma más rápida.

- **Capa de HUD e Interfaz:** Esta capa se renderiza siempre en la parte superior de la pantalla. No importa cuánto se mueva el coche o la cámara, el contador de diamantes siempre estarán fijos frente al jugador.

2.4.3 Componente de Objetos: Mecánicas y Coleccionables

Este apartado detalla los elementos interactivos que dan sentido a la jugabilidad y permiten la progresión del usuario. Se han diseñado siguiendo una arquitectura modular, lo que permite tratarlos como piezas independientes que se pueden colocar y configurar.

A. El Objeto Diamante (Sistema de Recolección)

El diamante es el principal incentivo para el jugador y se ha implementado como una escena independiente reutilizable. Este diseño es clave para la eficiencia del proyecto, ya que cualquier cambio en el diseño o la lógica del diamante se aplica instantáneamente a todos los diamantes repartidos por los distintos mundos.

- **Aspecto Visual y Animación:** Para que el objeto sea atractivo y destaque sobre los Tiles de hielo y tierra, se utiliza un nodo AnimatedSprite2D. Este nodo gestiona una animación en bucle que otorga brillo y movimiento constante al diamante, captando la atención del jugador incluso en zonas oscuras como los túneles.

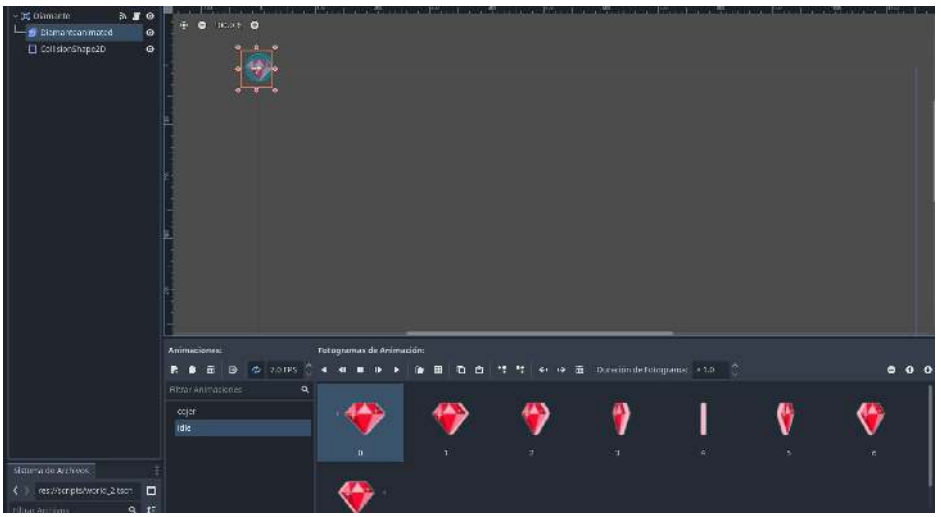


Foto de la animación del sprite Diamante

- **Lógica de Detección:** El "cuerpo" físico del diamante es un nodo Area2D equipado con una colisión circular (CollisionShape2D). Hemos ajustado el radio de esta colisión para que sea lo suficientemente generoso como para que el contacto con el coche se sienta justo y fluido.

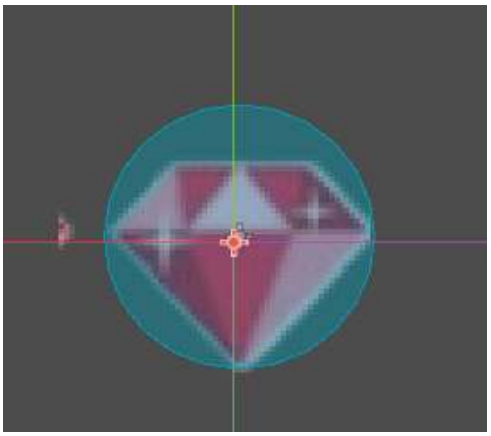


Foto del Diamante y su colisión (Área 2D)

- **Interfaz de Usuario:** El Contador (Label): Dentro de la arquitectura de la interfaz (HUD), el contador se define físicamente como un nodo de tipo Label.

  Este componente está diseñado para permanecer fijo en pantalla mediante un nodo CanvasLayer, lo que permite que el texto sea visible en todo momento sin importar el movimiento de la cámara o el desplazamiento del coche por el nivel. Su diseño visual está coordinado con el estilo del juego para que el número de diamantes recogidos sea legible sobre el fondo de los mundos.

Foto del Contador de Diamantes

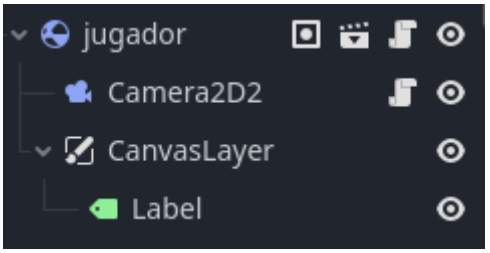


Foto de la jerarquía del nodo jugador

2.4.4 Punto de Control Final: Meta y Nodo Finish

El sistema de finalización de nivel no es simplemente una imagen decorativa al final del camino; es un componente técnico fundamental que hemos denominado Nodo Finish. Su función principal es actuar como un sensor que detecta cuándo el jugador ha superado todos los obstáculos y, de manera inmediata, activa el protocolo de victoria.

A. La Estructura del Nodo Area2D

Para la meta, hemos utilizado un nodo de tipo Area2D. A diferencia de los nodos que usamos para el suelo (que son sólidos y no dejan pasar al coche), el Area2D permite que el vehículo entre en su espacio físico sin detenerse bruscamente, permitiendo que el motor detecte la "superposición" de los dos cuerpos.

Dentro de este nodo, el elemento clave es la “CollisionShape2D”:

- **Forma y Posicionamiento:** Se ha configurado con una forma rectangular que abarca toda la altura de la pista. Esto es crucial por diseño: si el jugador llega a la meta con mucha velocidad y da un salto justo al final, el área debe ser lo suficientemente alta para capturarlo en el aire. No importa si el coche cruza por el suelo o volando, la colisión se detectará con precisión milimétrica.

- **Capas de Colisión:** Hemos configurado este nodo para que solo "escuche" al coche. Esto evita que otros elementos del escenario o efectos visuales disparen el final del nivel por error.

Foto del Área 2D y el sprite Finish

B. Estética y Feedback Visual

Para que el jugador identifique claramente hacia dónde debe ir, el nodo Finish incluye un componente visual (un Sprite o una bandera). Se ha colocado de forma que destaque sobre los colores del hielo y la tierra del escenario. Al combinarse con el área de colisión invisible, pero efectiva, se logra que el cruce de la meta sea un momento satisfactorio y sin errores técnicos, asegurando que el paso de un mundo a otro sea automático y fluido para el usuario.

Foto de la meta Finish

2.4.5 Interfaces de Usuario

El sistema de interfaces se ha diseñado siguiendo una estructura jerárquica de escenas, donde cada menú gestiona sus propios recursos de audio y lógica de navegación.

A. Interfaz Principal (Menú de Inicio)

Es la puerta de entrada al juego. Su objetivo es sumergir al jugador en la estética de RampTruck.

- **Contenido Visual:** Incluye un nodo VideoStreamPlayer que reproduce el vídeo de introducción en bucle, proporcionando dinamismo al fondo.
- **Jerarquía de Botones:**
    - **Botón "Jugar":** Realiza una transición mediante change_scene_to_file hacia la Interfaz de Niveles.
    - **Botón "Opciones/Salir":** Permite gestionar ajustes básicos o cerrar la aplicación.
- **Lógica:** Al entrar en esta escena, se detiene cualquier música de gameplay para dar prioridad al audio del vídeo promocional.

Foto de la Interfaz Principal

B. Interfaz de Selección de Niveles

Gestiona la progresión del jugador a través de los 5 mundos diseñados.

- **Estructura:** Se compone de un contenedor VBoxContainer con 5 botones interactivos.
- **Lógica de Desbloqueo:** Inicialmente, solo el Mundo 1 está disponible.

- Mediante un sistema de variables globales (GlobalProgress), el botón de cada mundo comprueba si el nivel anterior ha sido completado.
- Si el nivel está bloqueado, el botón se muestra con un icono de candado, impidiendo el acceso hasta que el jugador cruce la meta del nivel previo.

Foto de la Interfaz de Selección de Niveles

C. Interfaz de Victoria Final

Esta interfaz aparece como una capa superpuesta (Overlay) cuando el jugador detecta la colisión con la Meta.

- **Feedback Inmediato:** Muestra un panel con el mensaje "TE HAS PASADO RAMPTRUCK"
- **Opciones de Usuario:**
  - **Botón "finalizar":** Cierra el juego ya que se lo ha pasado

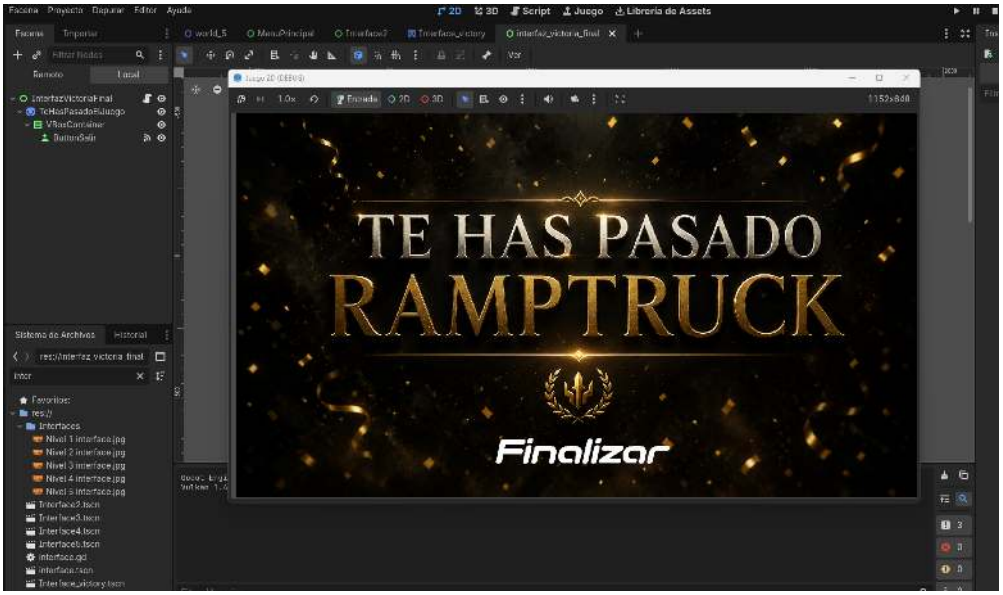


Foto de la Interfaz de Victoria

2.4.6 Trampas y peligros del escenario (Descripción)

Trampa de área: Pinchos se basan en un nodo Area2D que contiene un Sprite2D (visual) y un CollisionShape2D ajustado milimétricamente a la silueta del objeto. Su función es puramente punitiva: castigar el error de precisión del jugador.

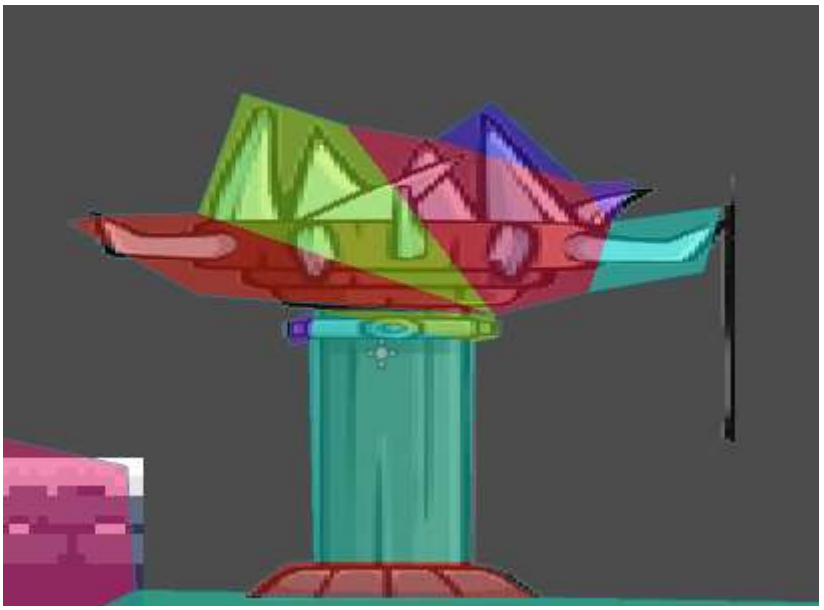


Foto de la Trampa 1 (pinchos)

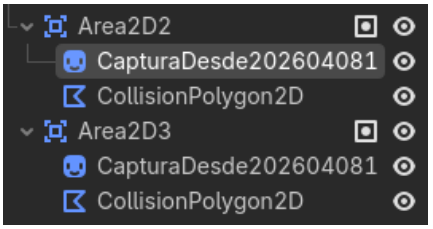


Foto de la jerarquía de las trampas de pincho

Foto de la Trampa 2 (pinchos)

Trampa Gigante: Prensa de aplastamiento esta es la trampa más compleja de los niveles avanzados. A diferencia de las anteriores, esta trampa tiene un ciclo de movimiento autónomo. Se configuró como un AnimatableBody2D. Esto es clave en Godot, ya que permite que un objeto se mueva por animación pero siga teniendo "cuerpo" físico que puede empujar o aplastar al jugador sin generar errores en el motor de físicas.

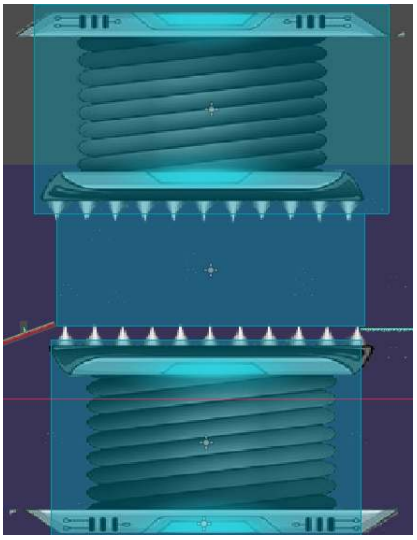


Foto de la Trampa Gigante

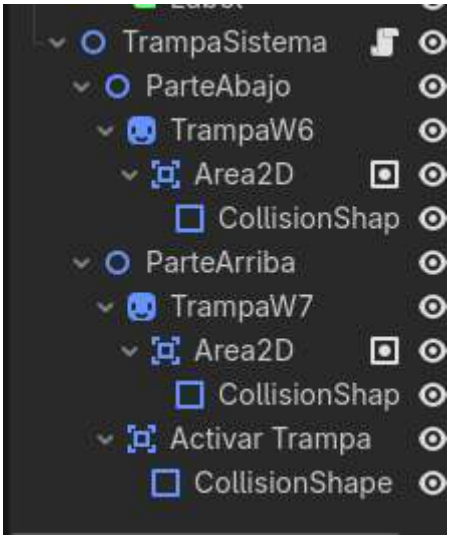


Foto de la estructura jerárquica de la Trampa Gigante

Trampa Carnívora Este componente ha sido diseñado como un obisatáculo dinámico que combina detección de proximidad y una animación reactiva de "ataque". Su arquitectura de nodos consta de:

- **AnimatedSprite2D:** Gestiona los estados visuales (Stand para espera, chomp para el ataque y tragar para la secuencia final).

- **AreaDeteccion (Area2D):** Un sensor de largo alcance que activa la animación de alerta cuando el coche se aproxima.

- **BocaColision (Area2D):** El área crítica. Si el coche entra en contacto con esta zona, se dispara la lógica de "muerte por devoración".

- **AudioStreamPlayer2D:** Nodo que reproduce un efecto sonoro de "mordisco" sincronizado exactamente con el frame de cierre de la boca.

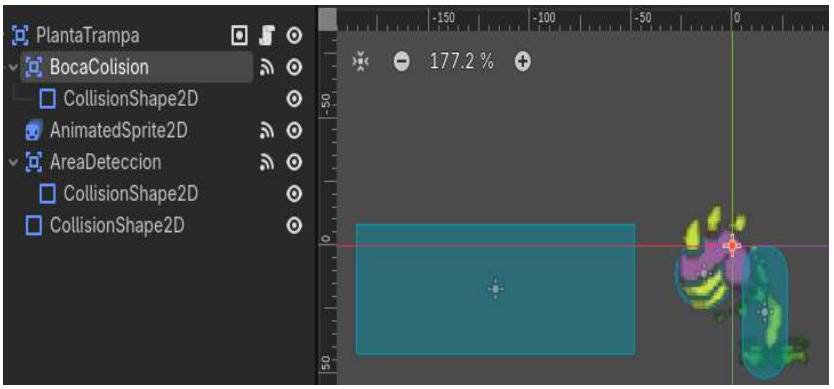


Foto de la estructura jerárquica de la Trampa Planta

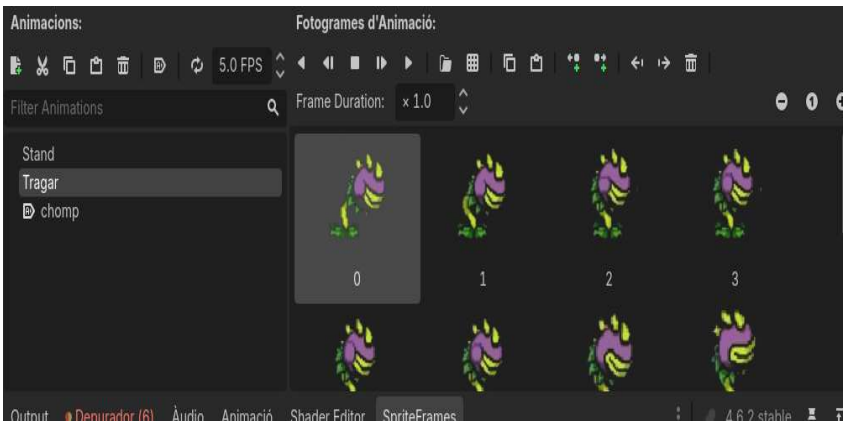


Foto de la animación del sprite Planta

2.4.7 Sistema de Teletransporte: Portales Vinculados

Este componente permite el desplazamiento instantáneo del vehículo entre dos coordenadas del mapa, añadiendo una mecánica de puzle y rapidez al diseño de niveles.

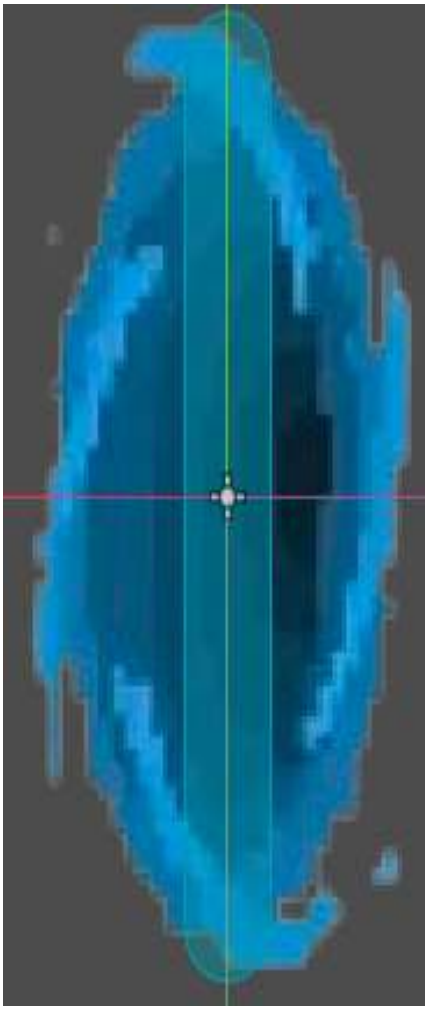


Foto del sprite de Portal

- **Arquitectura de Parejas**: Cada portal funciona como un objeto independiente, pero están diseñados para trabajar en parejas (Entrada y Salida).

- **Nodo Area2D y Marcador**: El portal utiliza un Area2D para detectar al coche y un nodo hijo llamado Marker2D (o un punto de destino) que indica la posición exacta donde debe aparecer el vehículo tras el transporte.

- **Variable Export (Target)**: Se utiliza una variable @export en el código. Esto permite que, desde el editor de Godot, podamos seleccionar manualmente a qué otro portal está conectado cada uno sin tener que escribir código nuevo para cada pareja.

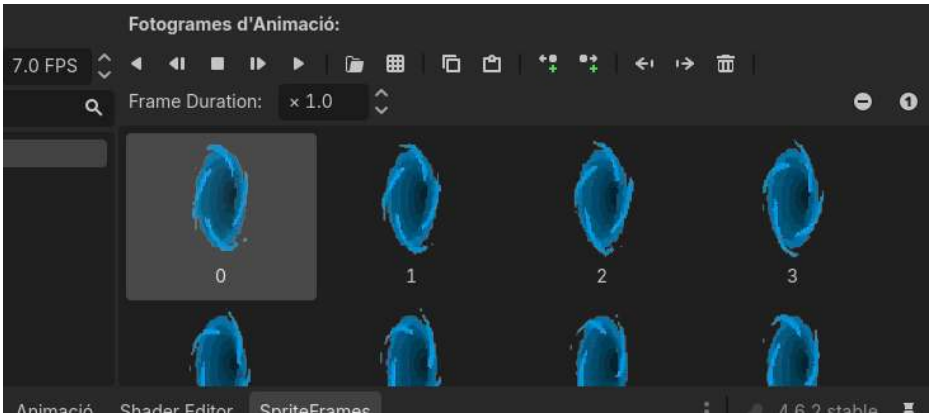


Foto de la animación del Portal

2.4.8 Sistema de Proyectiles: El Misil

El objeto se ha estructurado utilizando un nodo raíz de tipo Area2D anclado a un AnimatedSprite para la representación visual y un CollisionShape2D (forma de cápsula) ajustado al cuerpo del proyectil.

- **Arquitectura de la Escena misil.tscn**: Como se observa en la captura de pantalla de la escena del misil, el objeto se ha estructurado utilizando un nodo raíz de tipo Area2D anclado a un AnimatedSprite para la representación visual y un CollisionShape2D (forma de cápsula) ajustado al cuerpo del proyectil.

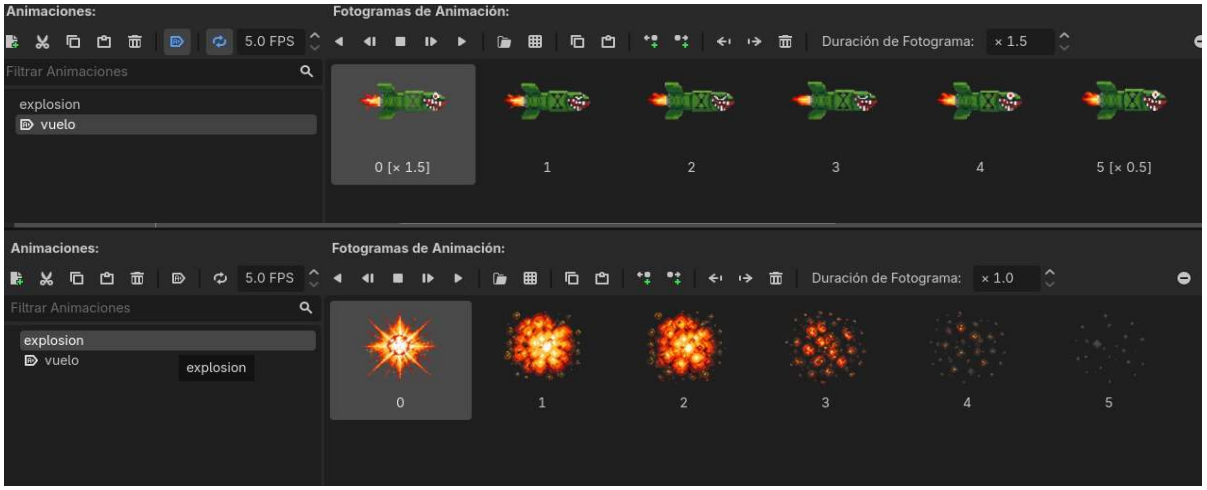


Foto de la animación del Misil cuando dispara y explota

Instanciación Dinámica: El vehículo utiliza el método instantiate() para crear copias de esta escena en tiempo real cada vez que el jugador pulsa la tecla de acción. El proyectil hereda la posición global del Marker2D frontal del coche y su rotación actual.

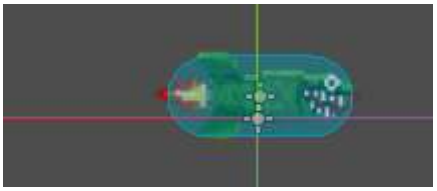


Foto del sprite Misil y su colisión (Área 2d)

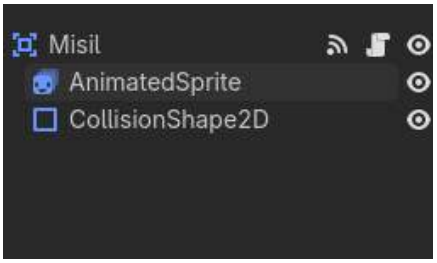


Foto de la estructura jerárquica del Misil

2.5 Definición de las funcionalidades

2.5.1 Programación Avanzada del Vehículo y Motor de Físicas (Teclas R, A, D)

A diferencia de un juego de plataformas simple, aquí hemos optado por un enfoque basado en la dinámica de cuerpos rígidos utilizando el nodo **RigidBody2D**. Esto implica que el coche no se mueve por coordenadas fijas, sino mediante la aplicación de fuerzas vectoriales, impulsos y torques.

A. Configuración de la Identidad Física (Masa y Gravedad):

- **Masa (mass = 20):** Al asignar un valor de 20, le otorgamos al vehículo una inercia considerable. Esto significa que al coche le cuesta empezar a moverse, pero una vez que alcanza velocidad, es más difícil detenerlo, fundamental para superar las rampas del Mundo 2 y 5.

- **Escala de Gravedad (gravity_scale = 5):** Hemos multiplicado la gravedad por cinco para evitar el efecto de "gravedad lunar". Esto obliga al jugador a ser muy preciso en los saltos.

B. Control mediante Teclas y Torque:

- **Teclas A y D:** No mueven el coche por coordenadas, sino que aplican aceleración a las ruedas y torque al chasis.

- **Torque en Suelo (500,000):** Cuando las ruedas tocan el suelo, el coche gira con menos fuerza para evitar volcar accidentalmente al subir una cuesta pequeña.

- **Torque en Aire (2,000,000):** En el aire, multiplicamos la fuerza por cuatro para permitir piruetas rápidas o corregir la trayectoria.

- **Tecla R:** Utilizada para reiniciar la escena si el jugador queda atrapado.

- **Función de Seguridad "Enderezar":** Aplica una fuerza masiva si el jugador pulsa la tecla correspondiente mientras está boca abajo. El código calcula el ángulo real del coche para saber si el techo está mirando al suelo.

C. El Sistema de Boost mediante "Doble Tap": Mediante el registro del tiempo (Time.get_ticks_msec()), el script detecta si el jugador pulsa la tecla derecha dos veces en menos de 0.25 segundos. Al activarse, el coche recibe un central_impulse constante durante 1.5 segundos en el eje local del coche (transform.x).

2.5.2 Lógica de Recolección, Señales (Diamante y Label) y Contador

Cuando el coche entra en el área del diamante, se dispara la señal body_entered. En ese instante, el sistema realiza tres acciones críticas:

1. Emite una señal hacia el HUD (Interfaz) para sumar los puntos correspondientes al contador global (Label).

2. Inicia una secuencia estética de desaparición para que el feedback visual sea satisfactorio.

3. Ejecuta una limpieza de memoria mediante la instrucción queue_free() para eliminar el objeto de la memoria y evitar que el ordenador se ralentice.

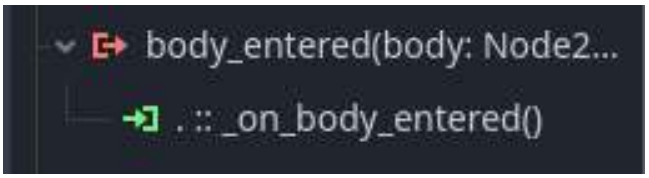


Foto de la señal conectada con la colisión

Funcionamiento del Contador y Actualización en Tiempo Real:

La lógica del contador no es estática, sino que depende directamente de las señales enviadas por los objetos de recolección.

- **Lógica de actualización:** El diamante, al ser recolectado, emite una señal hacia la interfaz. El script del Contador (Label) "escucha" esta señal y ejecuta una función que suma una unidad al valor actual de la variable.

- **Sincronización:** Para que el jugador vea el progreso, el script transforma ese valor numérico en texto mediante código, actualizando la propiedad `text` del nodo Label de forma instantánea. Este proceso asegura que no haya retraso entre el momento en que el coche toca el diamante y el momento en que el número cambia en la pantalla, ofreciendo un feedback inmediato al usuario.

face4 world_3 world_4 world_2 world_5 Carroc

a Depurar

```gdscript
extends Area2D

@onready var anim = $Diamanteanimated
func _ready():
    anim.play("idle")

func _on_body_entered(body):
    if body.is_in_group("jugador"):
        print("Diamante 1")
        anim.play("cojer")
        await (anim.animation_finished)
        Contador.Diamantes += 1
        queue_free()
    pass

```

Foto del Script del Diamante

2.5.3 Lógica de la Señal “body_entered” en la Meta

Cuando el coche entra en la meta, se dispara el evento “body_entered” conectado a un script que ejecuta:

- **Detención de Físicas:** El script envía una orden al coche para ignorar las entradas del teclado para que el jugador no siga acelerando durante la pantalla de victoria.

- **Activación de la Interfaz:** Llama automáticamente a la interface siguiente con el nivel desbloqueado.

- **Gestión de la Progresión:** La meta actúa como un "interruptor". Al tocarla, se envía una variable al sistema global indicando que ese mundo ha sido completado, permitiendo que en el menú aparezca el escudo dorado desbloqueado.

Foto de la Señal la cual está conectado el Área2d (body_entered)

Script el cual te envía a la interface siguiente cuando tocas el body entered (o sea cuando pasas la línea de meta)

```
5
6  func _on_area_2d_body_entered_world_3(_body: Node2D) -> void:
7      get_tree().change_scene_to_file("res://Interface4.tscn")
8
```

Foto de la línea de Script (body_entered)

2.5.4 Funcionamiento de la Cámara Dinámica y Seguridad de Zoom

La cámara en Rump Truck no es estática, sino que reacciona a las acciones del jugador para enfatizar la sensación de velocidad.

- **Zoom Progresivo mediante LERP:** Se ha implementado una función _process que calcula el zoom basándose en la entrada de usuario. Si el jugador acelera, la cámara realiza una interpolación lineal (lerp) hacia un valor de zoom más lejano (0.4), permitiendo ver más terreno por delante.

- **Llave de Seguridad (Variable vivo):** Uno de los retos técnicos fue evitar que la cámara siguiera haciendo zoom si el jugador mantenía pulsada la tecla tras morir. Para ello, se añadió una variable booleana vivo en el script del coche.

- **Sincronización Coche-Cámara:** La cámara ahora consulta el estado del grupo "jugador". Si detecta que el coche ha sido destruido o devorado (vivo = false), bloquea cualquier cambio de zoom y regresa automáticamente a la configuración normal (0.6), garantizando que la pantalla de reinicio se vea siempre de forma correcta y centrada.

```gdscript
func _process(delta):
    # Intentamos obtener el coche si aún no lo tenemos (por si aparece después)
    if not is_instance_valid(coche):
        _obtener_coche()

    var destino = zoom_normal
    var puede_hacer_zoom = false

    # Verificamos si el coche existe y tiene la variable 'vivo'
    if coche and "vivo" in coche and coche.vivo:
        puede_hacer_zoom = true

    # Lógica de decisión del destino
    if puede_hacer_zoom and Input.is_action_pressed("ui_right"):
        destino = zoom_acelerar
    else:
        destino = zoom_normal

    # Aplicamos el suavizado
    zoom = zoom.lerp(destino, velocidad_zoom * delta)
```

Foto del Script de la Configuración de la Cámara

2.5.5 Lógica de las Trampas Estáticas (Pinchos)

- **Funcionamiento**: Utilizan la señal body_entered. Al detectar el chasis del coche, llaman inmediatamente a la función explotar() del script del jugador.

- **Optimización**: Para evitar procesar colisiones innecesarias, estas trampas están configuradas en una capa de colisión específica (Layer 3), de modo que solo interactúan con el "Detector de Muerte" del coche y no con el suelo o elementos decorativos.

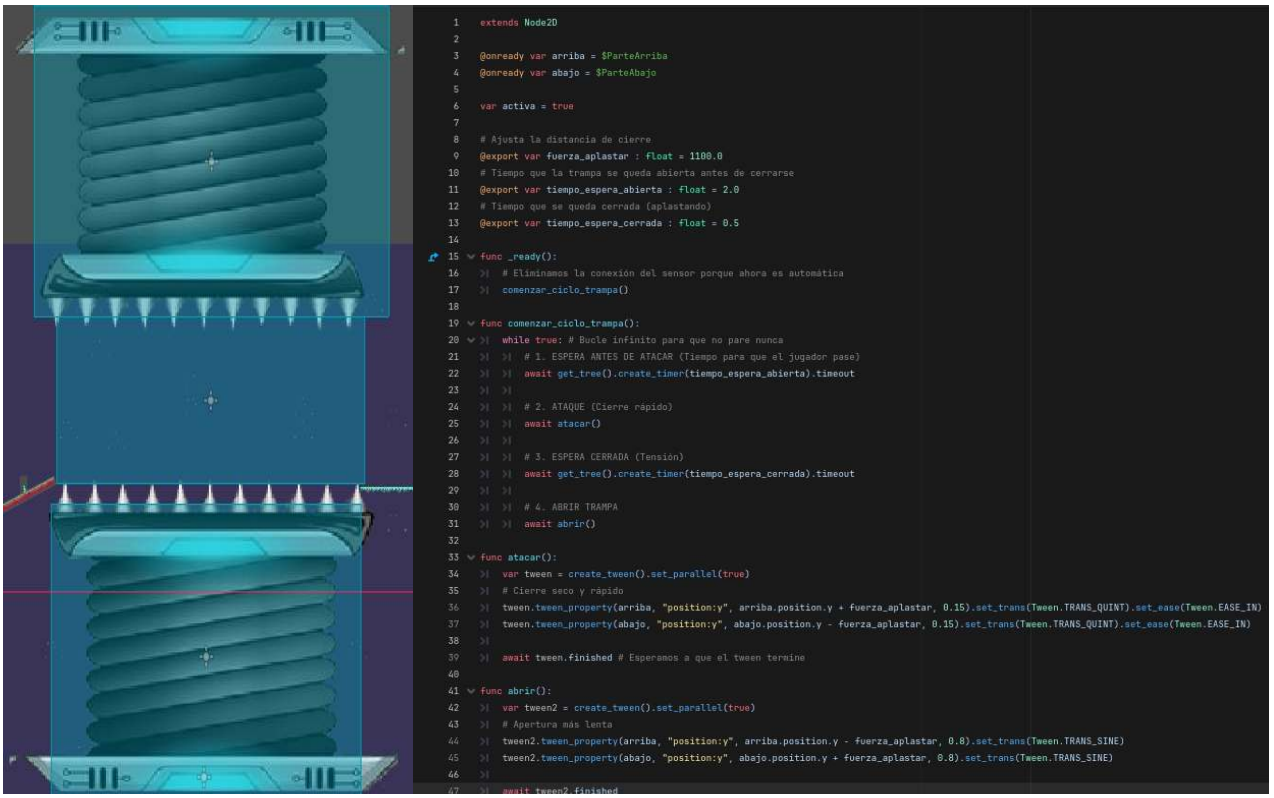
```
func _on_detector_de_muerte_area_entered(area: Area2D) -> void:
    if not vivo: return
    var nombre = area.name.to_lower()

    # Filtros de colisión de Script 2
    if "deteccion" in nombre or "activar" in nombre or "diamante" in nombre:
        return

    if area.name == "BocaColision":
        ser_devorado()
        return

    explotar()
```

Foto del Script de la Configuración del Detector de Muerte

Este algoritmo utiliza un sistema de exclusión. Al tocar un pincho del Mundo 2, el nombre del objeto no coincide con la "lista blanca" (diamantes o sensores), por lo que el script ejecuta automáticamente la función explotar().

2.5.6 Script y movimiento de la Trampa Mecánica (Mundo 5)

No se mueve mediante código de físicas pesado, sino mediante un nodo AnimationPlayer. Esto permite definir una curva de movimiento exacta: una bajada rápida y contundente (aplastamiento) y una subida lenta (recuperación).



```gdscript
extends Node2D

@onready var arriba = $ParteArriba
@onready var abajo = $ParteAbajo

var activa = true

# Ajusta la distancia de cierre
@export var fuerza_aplastar : float = 1100.0
# Tiempo que la trampa se queda abierta antes de cerrarse
@export var tiempo_espera_abierta : float = 2.0
# Tiempo que se queda cerrada (aplastando)
@export var tiempo_espera_cerrada : float = 0.5

func _ready():
    # Eliminamos la conexión del sensor porque ahora es automática
    comenzar_ciclo_trampa()

func comenzar_ciclo_trampa():
    while true: # Bucle infinito para que no pare nunca
        # 1. ESPERA ANTES DE ATACAR (Tiempo para que el jugador pase)
        await get_tree().create_timer(tiempo_espera_abierta).timeout

        # 2. ATAQUE (Cierre rápido)
        await atacar()

        # 3. ESPERA CERRADA (Tensión)
        await get_tree().create_timer(tiempo_espera_cerrada).timeout

        # 4. ABRIR TRAMPA
        await abrir()

func atacar():
    var tween = create_tween().set_parallel(true)
    # Cierre seco y rápido
    tween.tween_property(arriba, "position:y", arriba.position.y + fuerza_aplastar, 0.15).set_trans(Tween.TRANS_QUINT).set_ease(Tween.EASE_IN)
    tween.tween_property(abajo, "position:y", abajo.position.y - fuerza_aplastar, 0.15).set_trans(Tween.TRANS_QUINT).set_ease(Tween.EASE_IN)

    await tween.finished # Esperamos a que el tween termine

func abrir():
    var tween2 = create_tween().set_parallel(true)
    # Apertura más lenta
    tween2.tween_property(arriba, "position:y", arriba.position.y - fuerza_aplastar, 0.8).set_trans(Tween.TRANS_SINE)
    tween2.tween_property(abajo, "position:y", abajo.position.y + fuerza_aplastar, 0.8).set_trans(Tween.TRANS_SINE)

    await tween2.finished
```

Foto de la Trampa Mecánica y su Script

- **Sincronización de Peligro y Screen Shake:** Esta señal activa un pequeño temblor de cámara (Screen Shake) que aumenta la sensación de peso y peligro de la trampa para el jugador.

- **Automatización del Ciclo mediante Corrutinas (await):** El uso de await get_tree().create_timer() permite que la trampa funcione de forma autónoma sin necesidad de scripts externos complejos.

2.5.7 Funcionamiento de la Trampa Carnívora y mecánica de "Ser Devorado"

Se ha implementado un sistema de señales para que, al detectar al jugador, la planta pase de un estado pasivo a uno agresivo. Para diversificar las consecuencias de derrota, se ha programado la fragmentación del vehículo:

- **Efecto de Fragmentación (EfectoMuerte):** En lugar de simplemente ocultar el coche, hemos creado una estructura de "trozos" (TrozoDelantero y TrozoTrasero). Mediante código, el chasis real se invisibiliza y se activan estos fragmentos, que son desplazados mediante el uso de Tweens.

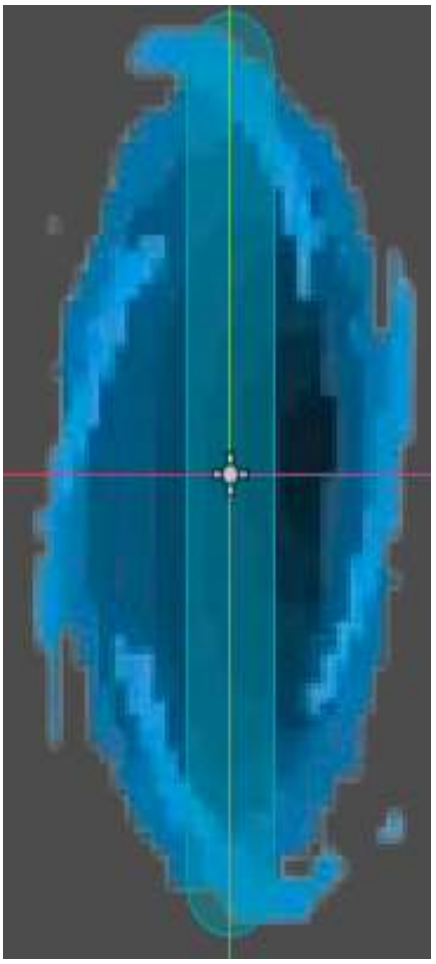
- **Uso de Tweens para la Escena Final:** Se ha utilizado la clase create_tween() para animar la posición de los restos del coche hacia el interior de la boca de la planta y, simultáneamente, reducir su opacidad (modulate:a) hasta desaparecer.

- **Bloqueo de Inercia:** Al activarse esta función, se fuerzan las variables linear_velocity y angular_velocity a cero. Esto es crítico para evitar que el coche "atraviese" la planta debido a la velocidad que llevaba antes de morir.

Foto de la Jerarquía, el Script y el Sprite del coche al ser devorado de la Trampa Planta

2.5.8 Lógica de Teletransporte y Reposicionamiento Físico

El teletransporte en un motor de físicas como Godot es complejo porque, si mueves un objeto bruscamente, el motor puede calcular colisiones erróneas. Por ello, se implementó la siguiente lógica:



- **Variable Export (Target):** Se utiliza una variable @export en el código. Esto permite que, desde el editor de Godot, podamos seleccionar manualmente a qué otro portal está conectado cada uno sin tener que escribir código nuevo para cada pareja.

- **Detección y Validación:** Cuando el coche entra en el Area2D, el script identifica el punto de destino vinculado.

- **Cambio de global_position:** Se actualiza la posición global del coche a la del portal de destino. El script permite elegir si el coche mantiene la velocidad que llevaba o si se resetea al aparecer, para mayor control del diseño del nivel. Para evitar que el coche entre en un "bucle infinito", se añade un pequeño retardo o se verifica que el coche no haya sido teletransportado recientemente.

```
func _on_body_entered(body):
    if body.name == "jugador" and not viajando:
        viajar(body)
```

Foto del Script de la teletransportación del Portal

2.5.9 Análisis Técnico del Proyectil (Script misil.gd)

Para que el sistema de disparo funcione, se ha desarrollado un script específico que gestiona la física y el ciclo de vida de cada proyectil:

```gdscript
extends Area2D

@export var velocidad = 800
var explotando = false

func _ready():
    # El tamaño medio que pediste
    scale = Vector2(5, 5)
    $AnimatedSprite.play("vuelo")

func _physics_process(delta):
    if not explotando:
        position += transform.x * velocidad * delta

# ESTA FUNCIÓN DETECTA EL SUELO Y PAREDES (StaticBody2D, TileMap)
func _on_body_entered(body):
    if explotando: return
    # Si toca cualquier cosa sólida, explota
    explotar()

# ESTA FUNCIÓN DETECTA LAS PLANTAS (Otras Area2D)
func _on_area_entered(area):
    if explotando: return

    # Si lo que toca es una planta o enemigo
    if "planta" in area.name.to_lower() or area.is_in_group("enemigos"):
        if area.has_method("morir"):
            area.morir()
        else:
            area.queue_free() # Si no tiene método morir, la borramos

        explotar()

func explotar():
    if explotando: return
    explotando = true
    velocidad = 0 # Se para en seco para la explosión
    $AnimatedSprite.play("explosion")

    await $AnimatedSprite.animation_finished
    queue_free()

```

Foto del script entero del Misil

- **Configuración de Movimiento:** En la función _physics_process(delta), calculamos una nueva posición multiplicando la dirección por la velocidad y el tiempo (delta). Esto garantiza que el misil se mueva de forma fluida independientemente de los FPS.

- **Gestión de Colisiones (_on_area_entered):** Cuando el Area2D del misil entra en contacto con otro cuerpo (como una prensa, un pincho o una caja), se activa esta función. Si el objeto impactado tiene una función de "daño" o "destrucción", el misil la activa. Tras el impacto, el misil ejecuta la instrucción queue_free().

- **Control de Cadencia (Cooldown):** Mediante el uso de corrutinas (await get_tree().create_timer), se ha limitado la velocidad de disparo del jugador.

- **Sistema de Autodestrucción por Tiempo:** Para evitar que un misil que no golpea nada siga viajando infinitamente, se incluye un temporizador (Nodo Timer). Al finalizar el tiempo (señal timeout), el misil se borra automáticamente.

2.5.10 Funcionamiento del Sonido

El sistema de audio del proyecto se ha diseñado para proporcionar un feedback inmediato y realista ante las acciones del jugador y los peligros del entorno. Se basa en tres pilares técnicos:

A. Sincronización mediante Señales de Animación (La Planta)

Para enemigos como la planta carnívora, el sonido no se reproduce de forma aleatoria, sino vinculado a la línea de tiempo visual:

- **Implementación:** Se utiliza el nodo AnimatedSprite2D. En el frame exacto donde la boca de la planta se cierra por completo, se dispara una señal (Signal) que ejecuta el método .play() del nodo AudioStreamPlayer2D.
- **Resultado:** Esto garantiza que el "mordisco" se escuche justo cuando ocurre el impacto visual, evitando la desincronización que arruinaría la inmersión.

B. Audio Espacial y Atenuación (2D vs Global)

Se han diferenciado dos tipos de emisores según su propósito:

- **Nodos AudioStreamPlayer2D:** Utilizados en elementos con posición fija en el mapa (plantas, portales, explosiones de misiles). El sonido se atenúa según la distancia a la cámara y permite el desplazamiento estéreo (si la planta está a la derecha, suena más por el auricular derecho).
- **Nodos AudioStreamPlayer (Global):** Utilizados para la música de fondo y los sonidos de victoria, asegurando un volumen constante independientemente de la posición del vehículo.

C. Control Dinámico del Motor y Turbo

El sonido del motor no es un archivo estático, sino que varía en tiempo real mediante script:

- **Pitch Shifting:** Al acelerar o usar el turbo, el script modifica la propiedad pitch_scale. Al aumentar la velocidad, el tono del audio se vuelve más agudo, simulando el aumento de revoluciones por minuto (RPM) del motor real.
- **Mezcla (Buses de Audio):** Todos los efectos pasan por un bus llamado "Moto" donde se ha aplicado una ligera distorsión para darle un carácter más agresivo y acorde a la estética del juego.

3. Otros Capítulos

Durante el desarrollo de RampTruck tuvimos que tomar varias decisiones importantes antes de empezar a programar. Para cada una de ellas, valoramos las opciones que teníamos en mente y elegimos la más adecuada según nuestro nivel y el tiempo del que disponíamos.

En un principio, nuestra intención era crear un videojuego en 3D (el del escape room) y después estuvimos decididos a diseñar el proyecto 2d pero ahora utilizando Roblox Studio. Sin embargo, tras las primeras pruebas y prototipos, nos dimos cuenta de que la complejidad de un entorno tridimensional era muy elevada para el tiempo que teníamos para entregar el proyecto. Además, la herramienta Roblox Studion no funcionaba correctamente en los ordenadores de clase, lo que nos generaba constantes problemas técnicos. Ante esta situación, el profesorado nos recomendó cambiar de enfoque y utilizar Godot Engine, una decisión que aceptamos y que resultó clave para garantizar que pudiéramos terminar el juego con éxito.

Al pasar a Godot, decidimos desarrollar el juego en 2D con una perspectiva lateral. Esta elección nos permitió centrarnos en lo que realmente queríamos que fuera el corazón de RampTruck: la física de la suspensión y el equilibrio del vehículo. Programar este comportamiento en dos dimensiones nos resultó mucho más manejable y nos permitió alcanzar una precisión técnica que en 3D habría sido casi imposible de pulir.

En cuanto al contenido, inicialmente pensamos en hacer un único recorrido largo, pero finalmente optamos por una estructura dividida en 5 Mundos temáticos. Esto nos permitió variar la estética visual utilizando diferentes Tilesets de hielo y tierra, además de aplicar una curva de dificultad progresiva para el jugador.

Sobre las mecánicas, valoramos la idea de añadir habilidades de salto convencionales, pero preferimos simplificar el sistema para dar más realismo al coche, centrándonos en el control del torque y el uso del boost. Para que el juego fuera más dinámico y divertido, decidimos añadir un sistema de disparo de misiles para eliminar obstáculos, como la planta carnívora, en lugar de implementar habilidades más complejas que habrían puesto en riesgo el funcionamiento del juego.

Para el almacenamiento y la gestión del proyecto, sabíamos que usaríamos GitHub. Esto nos permitió organizarnos mucho mejor como equipo, pudiendo trabajar de forma conjunta y guardar todos nuestros avances en los scripts y escenas de manera segura. Finalmente, decidimos que al perder o volcar el coche se reiniciara la escena, obligando al jugador a perfeccionar su conducción. Todas estas decisiones nos han ayudado a crear un juego sencillo, funcional y totalmente acorde al tiempo que teníamos disponible.

4. Conclusiones

4.1 Conclusiones generales del proyecto

Al haber desarrollado este videojuego a lo largo de todo este curso, hemos podido aprender conocimientos bastante importantes a la hora de crear un proyecto interactivo. Este proceso nos ha ayudado a entender mucho mejor el motor gráfico Godot Engine y a saber desarrollar videojuegos en 2D con un enfoque técnico en las físicas. Estos conocimientos nos pueden servir de cara a un futuro, por si nos gustaría dedicarnos profesionalmente a la creación de videojuegos, ya que anteriormente habíamos diseñado en 3D usando Unreal Engine y esta experiencia en 2D ha completado nuestra formación técnica.

4.2 Conseguimiento de los objetivos

A continuación, detallamos el estado de los objetivos marcados al inicio del proyecto:

- Diseñar la mecánica principal de movimiento del vehículo y la suspensión: (Hecho)
- Implementar un sistema de niveles funcional (5 Mundos): (Hecho)
- Desarrollar obstáculos dinámicos como la planta carnívora y trampas mecánicas: (Hecho)
- Implementar el sistema de disparo de misiles para la eliminación de enemigos: (Hecho)
- Crear una interfaz de usuario completa (Menú, Selección de niveles y Victoria): (Hecho)
- Documentar correctamente todo el proceso de desarrollo en esta memoria: (Hecho)

4.3 Valoración de la metodología y planificación

El desarrollo del proyecto se ha llevado a cabo de forma bastante estructurada. En la fase inicial establecimos las mecánicas principales que deberían aparecer en RampTruck, y a partir de ahí fuimos construyendo el juego paso a paso. Comenzamos por el sistema de movimiento del coche y su compleja suspensión, seguidos de la lógica de los enemigos y obstáculos, posteriormente la interfaz de usuario con el contador de diamantes y, por último, el diseño detallado de los niveles y los 5 mundos.

Durante el proceso fue necesario realizar algunos cambios y ajustes técnicos, como por ejemplo al equilibrar la dificultad de las rampas, ajustar la escala de gravedad o al incorporar nuevos peligros como los portales y las prensas. Estas modificaciones no supusieron un inconveniente, sino que forman parte habitual del desarrollo de un videojuego para asegurar que la experiencia sea divertida y desafiante.

La metodología utilizada resultó adecuada para un proyecto de estas características, ya que nos permitió avanzar de forma progresiva y facilitó la detección de errores en fases tempranas, especialmente en la integración de las colisiones manuales y el comportamiento del RigidBody2D.

5. Glosario

Await: Palabra clave en programación que sirve para que el código espere a que termine un proceso (como un temporizador) antes de seguir con la siguiente instrucción.

Body_entered: Es una "señal" que emite un nodo cuando un cuerpo físico (como el coche) entra en contacto con él. Es el disparador de la mayoría de lógicas del juego.

Instantiate (Instanciar): Proceso de crear una copia de una escena (como un misil) en tiempo real mientras el juego se está ejecutando.

Marker2D: Un punto invisible en el espacio que sirve como referencia para saber dónde debe aparecer algo (como el destino de un portal o el punto de salida de un misil).

Parallax Background: Efecto visual donde las capas del fondo se mueven a distintas velocidades para simular profundidad en un entorno 2D.

Queue_free(): Instrucción de código que borra un objeto del juego de forma segura, liberando la memoria RAM del ordenador para evitar que el juego se ralentice.

Torque: Fuerza de giro que se aplica a un cuerpo físico para hacerlo rotar sobre su eje, fundamental para el equilibrio del coche en el aire.

Tween: Herramienta de código que facilita la creación de animaciones sencillas de propiedades (como mover un objeto o cambiar su transparencia) sin usar el editor de animaciones.

6. Bibliografia

1. **Documentación Oficial de Godot**: https://docs.godotengine.org/en/stable/
2. **Coche y Ruedas***: https://www.youtube.com/watch?v=bAobnoBDD4I*
   Data: 24/11/25 - 3/12/25
3. **Contador**: https://www.youtube.com/watch?v=_3LWmPAiGl0
   Data: 18/3/26
4. **Aprendizaje general:** https://www.youtube.com/watch?v=Wa4yO92SXkc y https://www.youtube.com/watch?v=jMhi5XDJ7VY
   Data: 8/12/25
5. **Tile Map:** https://www.youtube.com/watch?v=A4zU3p1EXVY
   Data: 8/12/25 - 7/1/26
6. **Parallax y Background**: https://www.youtube.com/watch?v=f8z4x6R7OSM&t=17s
   Data: 8/12/25 - 7/1/26
7. **Trampas**: https://www.youtube.com/watch?v=JikBRPK7p2k
   Data: 20/4/25
8. **Diamante**: https://www.youtube.com/watch?v=9hRT0W4bjB4
   Data: 25/2/25
9. **Portal**: https://www.youtube.com/watch?v=YT5ZE0a4YtU
   Data: 22/4/25
10. **Interfaz y botones:** https://www.youtube.com/watch?v=55U-hnUnFAY y https://www.youtube.com/watch?v=5Hog6a0EYa0
    Data: 7/1/26
11. **Assets de Sonido**: https://freesound.org/ (Licencia Creative Commons).
12. **Sprites y Texturas**: https://kenney.nl/assets (Assets gratuitos de uso público).
13. **Comunidad Godot España (google)**: Consultas sobre señales y optimización de código.

Las fechas de desarrollo han sido extraídas del documento de seguimiento semanal del proyecto, https://docs.google.com/spreadsheets/d/166iVxa5DSNKljB1tqgcPu1yM4fJ7Xtc4ZcKpK3wgu7M/edit?gid=0#gid=0

7. Annexos

```gdscript
extends Area2D

@onready var anim = $Diamanteanimated
func _ready():
    anim.play("idle")

func _on_body_entered(body):
    if body.is_in_group("jugador"):
        print("Diamante 1")
        anim.play("cojer")
        await (anim.animation_finished)
        Contador.Diamantes += 1
        queue_free()
    pass

```

Foto del Script Diamante (recolectar + contador)

```gdscript
extends Control

func _on_button_pressed() -> void:
    get_tree().change_scene_to_file("res://scripts/world.tscn")


func _on_button_2_pressed() -> void:
    get_tree().change_scene_to_file("res://scripts/world_2.tscn")


func _on_button_3_pressed() -> void:
    get_tree().change_scene_to_file("res://scripts/world_3.tscn")


func _on_button_4_pressed() -> void:
    get_tree().change_scene_to_file("res://scripts/world_4.tscn")


func _on_button_5_pressed() -> void:
    get_tree().change_scene_to_file("res://scripts/world_5.tscn")

```

Foto del Script cambiar de interfaz al pulsar los botones

```
1   extends Control
2
3
4   func _on_button_salir_pressed() -> void:
5       get_tree().quit()
6
```

Foto del Script de botón de interfaz de victoria

```
1   extends RigidBody2D
2
3   # ====================
4   # VARIABLES DE DISPARO
5   # ====================
6   @export var misil_escena = preload("res://misil.tscn")
7   var puede_disparar = true
8
9   @onready var punto_disparo = get_node_or_null("CarBody/PuntoDisparo")
10  # --- SISTEMA DE AUDIO ---
11  @onready var sfx_explosion = $sfx_explosion  # Añade esta línea arriba
12  @onready var sfx_estatico = $estatico
13  @onready var sfx_acceleracion = $acceleracion
14  @onready var sfx_boost = $accelerar2
15  var ya_esta_sonando = false
```

Foto del Script del Jugador variable de disparo + sonido

```
18    # VARIABLES DE MOVIMIENTO
19    # ======================
20    var ruedas = []
21    var vivo = true
22    var velocidad = 120000
23    var torque_aire = 1800000
24    var torque_suelo = 500000
25    var fuerza_enderezar = 4000000000
26
27    # 👉 BOOST (Doble Tap original)
28    var boost_activo = false
29    var tiempo_boost = 0.0
30    var duracion_boost = 1.5
31    var fuerza_boost = 240000
32
33    # 👉 DOBLE TAP
34    var ultimo_tap = 0.0
35    var ventana_tap = 0.25
```

Foto de la variable de movimiento (para controlar el coche)

```
38  # READY
39  # =====================
40  # Mecanica de sonido, gravedad y masa, ruedas
41  func _ready():
42      if sfx_estatico:
43          sfx_estatico.play()
44          sfx_estatico.volume_db = 0.0 # Que se escuche el motor encendido
45      mass = 17
46      gravity_scale = 7
47      angular_damp = 3.0
48
49      if has_node("RuedasUnidas1"): ruedas.append($RuedasUnidas1)
50      if has_node("RuedasUnidas2"): ruedas.append($RuedasUnidas2)
51
52      for r in ruedas:
53          if r is RigidBody2D:
54              r.can_sleep = false
55              r.add_collision_exception_with(self)
```

Foto de la Configuración del sonido, gravedad y masa

```gdscript
# PHYSICS (Movimiento y Disparo)
# ====================
func _physics_process(delta):
    if freeze or not vivo: return

    if Input.is_action_just_pressed("ui_accept") and puede_disparar:
        disparar_misil()

    var right_pressed = Input.is_action_just_pressed("ui_right")
    var right_hold = Input.is_action_pressed("ui_right")
    var left = Input.is_action_pressed("ui_left")

    if right_pressed:
        var ahora = Time.get_ticks_msec() / 1000.0
        if ahora - ultimo_tap < ventana_tap:
            _activar_boost()
        ultimo_tap = ahora

    if boost_activo:
        tiempo_boost -= delta
        if tiempo_boost <= 0: boost_activo = false

    if right_hold:
        for r in ruedas:
            r.apply_torque_impulse(velocidad * delta * 60)
        if boost_activo:
            var fuerza_dir = Vector2(transform.x.x, 0).normalized()
            apply_central_impulse(fuerza_dir * fuerza_boost * delta)
    elif left:
        for r in ruedas:
            r.apply_torque_impulse(-velocidad * delta * 60)

    if right_hold:
        apply_torque_impulse((torque_suelo if _en_suelo() else torque_aire) * delta)
    if left:
        apply_torque_impulse(-(torque_suelo if _en_suelo() else torque_aire) * delta)
```

Foto del Movimiento del coche (botones, torque suelo y aire mas boost)

```gdscript
# ====================
# SISTEMA DE MUERTE
# ====================
func _on_detector_de_muerte_area_entered(area: Area2D) -> void:
    if not vivo: return
    var nombre = area.name.to_lower()

    if "deteccion" in nombre or "activar" in nombre or "diamante" in nombre or "portal" in nombre or "punto" in nombre:
        return

    if area.name == "BocaColision":
        ser_devorado(area.global_position)
        return

    explotar()

func explotar():
    if not vivo: return
    var pos_muerte = global_position
```

Foto del Sistema de muerte (explotar si toca algo…)

```gdscript
if input_acel:
    if not ya_esta_sonando:
        sfx_acceleracion.play()
        sfx_boost.play()
        ya_esta_sonando = true

    if boost_activo:
        # --- ESTADO TURBO ---
        sfx_boost.volume_db = lerp(sfx_boost.volume_db, 6.0, 0.1)
        sfx_acceleracion.volume_db = lerp(sfx_acceleracion.volume_db, -30.0, 0.1)

        # No subas de 1.4 o 1.5, si no, suena a juguete
        sfx_boost.pitch_scale = lerp(sfx_boost.pitch_scale, 1.45, 0.05)
    else:
        # --- ESTADO NORMAL ---
        sfx_boost.volume_db = lerp(sfx_boost.volume_db, -40.0, 0.1)
        sfx_acceleracion.volume_db = lerp(sfx_acceleracion.volume_db, 2.0, 0.05)
        sfx_acceleracion.pitch_scale = lerp(sfx_acceleracion.pitch_scale, 1.15, 0.02)

        # Tono de motor normal
        sfx_acceleracion.pitch_scale = lerp(sfx_acceleracion.pitch_scale, 1.1, 0.02)
else:
    # --- ESTADO PARADO ---
    sfx_boost.volume_db = lerp(sfx_boost.volume_db, -45.0, 0.05)
    sfx_acceleracion.volume_db = lerp(sfx_acceleracion.volume_db, -45.0, 0.05)
    sfx_estatico.volume_db = lerp(sfx_estatico.volume_db, 0.0, 0.05)

    if ya_esta_sonando and sfx_acceleracion.volume_db < -35:
        sfx_acceleracion.stop()
        sfx_boost.stop()
        ya_esta_sonando = false
```

Foto de la Configuración del sonido (volumen y duración cuando pulsas el boost)

```gdscript
extends Control

func _on_button_pressed() -> void:
	get_tree().change_scene_to_file("res://interface.tscn")


func _on_button_salir_pressed() -> void:
	get_tree().quit()

```

Foto del Script del Menú principal

```gdscript
extends Area2D

@export var velocidad = 800
var explotando = false

# --- REFERENCIA AL SONIDO ---
@onready var sfx_explosion = $sfx_explosion_misil

func _ready():
	scale = Vector2(5, 5)
	$AnimatedSprite.play("vuelo")

func _physics_process(delta):
	if not explotando:
		position += transform.x * velocidad * delta

# DETECTA SUELO Y PAREDES
func _on_body_entered(body):
	if explotando: return
	# Ignoramos al jugador para que no explote en nuestra cara
	if body.name.to_lower().contains("jugador"): return
	explotar()

# DETECTA PLANTAS/ENEMIGOS
func _on_area_entered(area):
	if explotando: return

	if "planta" in area.name.to_lower() or area.is_in_group("enemigos"):
		if area.has_method("morir"):
			area.morir()
		else:
			area.queue_free()

```

Foto del Script del Misil

```gdscript
func explotar():
    if explotando: return
    explotando = true

    # 1. Lógica física: se para y deja de detectar choques
    velocidad = 0
    set_deferred("monitoring", false)
    set_deferred("monitorable", false)

    # 2. Lógica visual: Animación
    $AnimatedSprite.play("explosion")

    # 3. Lógica de Audio:
    if sfx_explosion:
        # Le damos un pitch aleatorio para que suene mejor (menos aspiradora)
        sfx_explosion.pitch_scale = randf_range(1.1, 1.3)
        sfx_explosion.play()

    # 4. Esperamos a que la animación termine
    await $AnimatedSprite.animation_finished

    # 5. Si el sonido sigue sonando, esperamos a que termine antes de borrar el misil
    if sfx_explosion and sfx_explosion.playing:
        visible = false # Lo escondemos para que parezca que ya se ha ido
        await sfx_explosion.finished

    queue_free()
```

Foto del Script del misil (mecánicas)

```gdscript
@onready var sprite = $AnimatedSprite2D
@onready var colision_boca = $BocaColision

func _ready():
	sprite.play("Stand")
	# La boca empieza desactivada para que no te mate antes de tiempo
	colision_boca.set_deferred("disabled", true)

# 1. El detector largo (Rectángulo azul) activa la animación
func _on_area_deteccion_body_entered(body):
	# Si lo que entra es un RigidBody2D (como tu coche), activamos
	if body is RigidBody2D:
		if sprite.animation != "chomp":
			print("¡PLANTA: Detectado cuerpo físico, muerde!")
			sprite.play("chomp")
	else:
		print("¡PLANTA: Algo entró pero no es el coche (es un: ", body.name, ")")

# 2. Control de los frames: la boca solo muerde cuando está abierta
func _on_animated_sprite_2d_animation_finished():
	if sprite.animation == "chomp":
		# Opción A: Vuelve a esperar (boca abierta)
		sprite.play("Stand")

# 3. Cuando el coche toca la BocaColision
# En el script de la planta (planta_trampa.gd)
func _on_boca_colision_body_entered(body):
	if body.has_method("ser_devorado"):
		# Le pasamos la posición de la planta para que la cámara sepa a dónde mirar
		body.ser_devorado(global_position)

		# Animación de tragar
		sprite.play("Tragar")

		# Esperas y respawn
		await sprite.animation_finished
		await get_tree().create_timer(0.3).timeout
		get_tree().reload_current_scene()
```

Foto del Script Planta Trampa

```gdscript
extends Node2D

func _on_button_2_pressed() -> void:
	get_tree().change_scene_to_file("res://interface.tscn")


func _process(delta: float) -> void:
	$jugador/CanvasLayer/Label.text = "Diamantes: " + str(Contador.Diamantes)

func _on_area_2d_body_entered(body: Node2D) -> void:
	get_tree().change_scene_to_file("res://Interface3.tscn")

```

Foto del Script de los mundos (misma jerarquía solo cambiando la interfaz)

```gdscript
extends Area2D

# Referencias a nodos hijos
@onready var punto_salida = $"Punto de salida"
@onready var sonido_teleport = $SonidoTP

# Variable de control local (cada portal tiene la suya)
var bloqueado = false

func _on_body_entered(body):
    # Verificamos que el nombre contenga "jugador" y que este portal no esté en cooldown
    if body.name.to_lower().contains("jugador") and not bloqueado:
        viajar(body)

func viajar(body):
    bloqueado = true

    # REPRODUCCION DEL SONIDO
    if sonido_teleport:
        sonido_teleport.play()

    # --- 1. ENTRADA AL PORTAL (Efecto de succión) ---
    if body is RigidBody2D:
        body.freeze = true

    var tween_in = create_tween().set_parallel(true)
    # Se encoge y se mueve al centro del portal actual
    tween_in.tween_property(body, "scale", Vector2(0, 0), 0.4).set_trans(Tween.TRANS_BACK).set_ease(Tween.EASE_IN)
    tween_in.tween_property(body, "global_position", global_position, 0.4)

    await tween_in.finished
```

```gdscript
    # --- 2. TELETRANSPORTE ---
    # Posicionamos al jugador en el destino
    body.global_position = punto_salida.global_position

    # --- 3. SALIDA (Efecto de aparición) ---
    var tween_out = create_tween()
    tween_out.tween_property(body, "scale", Vector2(1, 1), 0.5).set_trans(Tween.TRANS_ELASTIC).set_ease(Tween.EASE_OUT)

    await tween_out.finished

    # --- 4. LANZAMIENTO (El "Escupitajo") ---
    if is_instance_valid(body) and body is RigidBody2D:
        body.freeze = false
        # Dirección basada en la rotación del Punto de Salida
        var direccion = Vector2.RIGHT.rotated(punto_salida.global_rotation)
        body.apply_central_impulse(direccion * 1200)

    # Cooldown de 1 segundo para este portal específico
    await get_tree().create_timer(1.0).timeout
    bloqueado = false
```

Foto del Script Portal (un poco de ia)

```gdscript
extends Node2D

@onready var arriba = $ParteArriba
@onready var abajo = $ParteAbajo

var activa = true

# Ajusta la distancia de cierre
@export var fuerza_aplastar : float = 1100.0
# Tiempo que la trampa se queda abierta antes de cerrarse
@export var tiempo_espera_abierta : float = 2.0
# Tiempo que se queda cerrada (aplastando)
@export var tiempo_espera_cerrada : float = 0.5

func _ready():
	# Eliminamos la conexión del sensor porque ahora es automática
	comenzar_ciclo_trampa()

func comenzar_ciclo_trampa():
	while true: # Bucle infinito para que no pare nunca
		# 1. ESPERA ANTES DE ATACAR (Tiempo para que el jugador pase)
		await get_tree().create_timer(tiempo_espera_abierta).timeout

		# 2. ATAQUE (Cierre rápido)
		await atacar()

		# 3. ESPERA CERRADA (Tensión)
		await get_tree().create_timer(tiempo_espera_cerrada).timeout

		# 4. ABRIR TRAMPA
		await abrir()
```

```gdscript
func atacar():
	var tween = create_tween().set_parallel(true)
	# Cierre seco y rápido
	tween.tween_property(arriba, "position:y", arriba.position.y + fuerza_aplastar, 0.15).set_trans(Tween.TRANS_QUINT).set_ease(Tween.EASE_IN)
	tween.tween_property(abajo, "position:y", abajo.position.y - fuerza_aplastar, 0.15).set_trans(Tween.TRANS_QUINT).set_ease(Tween.EASE_IN)

	await tween.finished # Esperamos a que el tween termine

func abrir():
	var tween2 = create_tween().set_parallel(true)
	# Apertura más lenta
	tween2.tween_property(arriba, "position:y", arriba.position.y - fuerza_aplastar, 0.8).set_trans(Tween.TRANS_SINE)
	tween2.tween_property(abajo, "position:y", abajo.position.y + fuerza_aplastar, 0.8).set_trans(Tween.TRANS_SINE)

	await tween2.finished
```

Foto del Script Trampa Gigante (un poco de ia)