laSalle

UNIVERSITAT RAMON LLULL

Escola Tècnica Superior d'Enginyeria La Salle

Treball Final de Grau

Grau en Enginyeria Multimèdia

Desarrollo de un videojuego para Android en Godot Engine

Alex Collado Garrido

Ferran Ruiz Sala

ACTA DE L'EXAMEN
DEL TREBALL FI DE CARRERA

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D. Alex Collado Garrido

va exposar el seu Treball de Fi de Carrera, el qual va tractar sobre el tema següent:

Desenvolupament d’un videojoc per Android en Godot Engine

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

Abstracto

Este proyecto consiste en el diseño y desarrollo de un videojuego para móviles con el sistema operativo *Android*. El juego ha sido desarrollado utilizando el motor gráfico Godot, y ha sido publicado en *itch.io*.

El juego pertenece a la categoría del árcade, que describe juegos con unas mecánicas simples y fáciles de aprender, pero difíciles de dominar, con partidas de corta duración y que se centran principalmente en su jugabilidad. Las partidas se inspiran en las máquinas de «*air hockey*», en las que hay que golpear un disco y hacer que rebote en las paredes para llevarlo a la portería rival.

El proyecto explicara todo el proceso de desarrollo de un videojuego multijugador para *Android* creado utilizando el motor gráfico Godot, cubriendo los apartados de diseño, técnicos y artísticos, entrando en detalle en la conceptualización, el diseño de las mecánicos, la programación, los desafíos técnicos del multijugador en linea, el acabado artístico tanto visual como sonoro y la usabilidad.

Durante los diferentes estados de madurez del proyecto se han realizado sesiones de prueba con varios jugadores de diferente bagaje en el mundo de los videojuegos para ayudar a detectar y resolver problemas y aportar diferentes puntos de vista para ayudar a mejor el proyecto.

Todo el proceso de desarrollo de este proyecto se ha llevado a cabo utilizando unicamente herramientas de código abierto, como el anteriormente mencionado Godot, Blender para todas las necesidades de modelado, GIMP y Inkscape para la edición de imágenes y la *suite* de LibreOffice para toda la redacción.

El objetivo final de este proyecto es crear un juego divertido y que se pueda disfrutar con otras personas mientras se aprende en el proceso.

Abstracte

Aquest projecte consisteix en el disseny i desenvolupament d'un videojoc per mòbils amb el sistema operatiu Android. El joc ha sigut desenvolupat utilitzant el motor gràfic Godot, i ha sigut publicat en *itch.io.*

El joc pertany a la categoria de l'àrcade, que descriví jocs amb unes mecàniques simples i fàcils d'aprener, però difícils de dominar, amb partides de duració curta i que se'n centren principalment en la seva jugabilitat. Les partides s'inspiren principalment en les màquines de «*air hockey*», en les quals s'ha de colpejar un disc i fer que reboti en les parts per portar-ho a la porteria rival.

El projecte explicarà tot el procés de desenvolupament d'un videojoc multijugador per a Android creat utilitzant el motor gràfic Godot, cobrint els apartats de disseny, tècnics i artístics, entrant en detall en la conceptualització, el disseny de les mecàniques, la programació, els reptes tècnics del multijugador en línia, l'acabat artístic tant visual com sonor i la usabilitat.

Durant els diferents estats de maduresa del projecte se n'han realitzat sessions de proves amb diversos jugadors de diferent bagatge al món dels videojocs per ajudar a detectar i resoldre problemes i aportar diferents punts de vista per ajudar a millorar millor el projecte.

Tot el procés de desenvolupament d'aquest projecte s'ha portat a terme utilitzant únicament eines de codi obert, com l'anteriorment esmentat Godot, Blender per a totes les necessitats de modelatge, GIMP i Inkscape per a l'edició d'imatges i la suite de LibreOffice per a tota la redacció.

L'objectiu final d'aquest projecte és crear un joc divertit i que se'n pugui gaudir amb altres persones m'entres s'aprèn en el procés.

Abstract

This project consists of the design and development of a video game for cell phones with the Android operating system. The game has been developed using the Godot graphics engine, and has been published in itch.io.

The game belongs to the arcade category, which describes games with simple and easy to learn, but difficult to master mechanics, with short duration games that focus mainly on gameplay. The games are inspired by “air field hockey” machines, in which you have to hit a puck and make it bounce off the walls to take it to the opponent's goal.

The project will explain the entire development process of a multiplayer video game for Android created using the Godot graphics engine, covering the design, technical and artistic sections, going into detail on the conceptualization, the design of the mechanics, programming, the technical challenges of online multiplayer, the artistic finish both visual and sound and usability.

During the different stages of maturity of the project, test sessions have been conducted with several players from different backgrounds in the world of video games to help detect and solve problems and provide different points of view to help improve the project.

The whole development process of this project has been carried out using only open source tools, such as the previously mentioned Godot, Blender for all modeling needs, GIMP and Inkscape for image editing and the LibreOffice suite for all the writing.

The ultimate goal of this project is to create a game that is fun and can be enjoyed with others while learning in the process.

Agradecimientos

Me gustaría agradecer a mi familia y amigos, por hacer tolerable no solo este trabajo, sino todo el proceso que ha llevado hasta el.

A mi tutor en este proyecto, Ferran Ruiz, por los buenos consejos que me ha dado para ayudarme durante el desarrollo.

A todas las personas que han ayudado a probar al juego y han dado su opinión.

Y al Cooper.

Contenido

Acrónimos

API: *Application Programming Interface*

FOSS: Free & Open Source Software

GPU: Graphical Processing Unit

GML: Game Makers Language

HTTP: Hypertext Transfer Protocol

HTTPS: *Hypertext Transfer Protocol Secure*

IA*:* Inteligencia Artificial

IP: Internet Protocol

IDE: Integrated Development Environment

ISP: Internet Services Provider

I+D: Investigación y desarrollo

JRPG: Japanese Role Playing Game

LAN: Local Area Network

NES: Nintendo Entertainment System

P2P: Peer to peer

RAM: Random Access Memory

RPC: *Remote procedure calls*

RPG: Role Playing Game

SCUMM*: Script Creation Utility for Maniac Mansion*

SDK: *Software Development Kit*

TCP: Transmission Control Protocol

TFG: Trabajo Final de Grado

UDP: *User Datagram Protocol*

1 Introducción

Desde un principio, el objetivo de este trabajo ha sido el desarrollo de un videojuego. Estaba decidido desde mucho antes de empezar con la elaboración de este proyecto. Pero el resto de variables en juego, como que tipo de juego sería, qué herramientas se usan para su creación, para que plataformas será desarrollado, se han ido decidiendo basándose en la investigación, diferentes pruebas y desarrollo general de este trabajo.

El camino que lleva a cada una de estas decisiones será detallado en los siguientes apartados de este documento, que están pensados para representar las diferentes fases de la confección de este proyecto. Se hablara tanto de las partes más técnicas del proceso como de las de diseño, y se expondrán los problemas que han surgido en estas y las diferentes soluciones que se han aplicado.

Este trabajo presenta mi experiencia personal con el desarrollo de este proyecto, pero también apunta a servir como guía para cualquier persona que quiera embarcarse en un proyecto similar en un futuro.

El propósito principal de este proyecto ha sido aprender a manejar un desarrollo en solitario y circunnavegar entre todas las ventajas y desventajas que esto supone mientras se busca producir un juego de principio a fin, que sea divertido de jugar y que no de la impresión de no estar acabado o de haberse quedado a medio camino.

2 Marco teórico

2.1 Motores gráficos

2.1.1 Introducción

Un motor gráfico es un *framework* de *software*[1] diseñado principalmente para el desarrollo de videojuegos que suele incluir bibliotecas y programas de apoyo pertinentes, como por ejemplo un editor de niveles, para ayudar a desarrollar y unir los diferentes componentes de un proyecto [1] . Estos son una pieza fundamental en el desarrollo de videojuegos, ya que liberan al equipo de desarrollo de una gran carga de trabajo.

El motor gráfico, al cual a partir de ahora vamos a referirnos como únicamente el motor, se encarga de las tareas de más bajo nivel[2], como realizar las llamadas a la API gráfica para renderizar gráficos por pantalla, traducir la entrada del teclado, ratón o mando en valores con los que se pueda trabajar y en general funciones que van a ahorrar grandes cantidades de tiempo y trabajo a las personas que estén utilizando el programa.

Las funciones y herramientas que ofrece un motor varían enormemente entre ellos y depende enteramente de la propuesta de cada motor. Hoy en día existe una enorme cantidad de motores abiertos al público general y muchos de ellos son muy concretos y especializados, estando diseñados para crear juegos de un género en concreto. Otros se centran en ser sencillos de usar y accesibles para usuarios que están empezando en el desarrollo de videojuegos o que no tienen experiencia en programación, a cambio de tener herramientas menos profundas y versátiles. Los más populares son los motores de uso general, que están pensados para ser lo más versátiles posible y poder usarse para crear cualquier juego imaginable, aunque suelen requerir experiencia técnica en diferentes ámbitos para poder exprimir todo su potencial.

En esta sección se repasará por encima la historia del los motores gráficos desde mediados de los 80 hasta día de hoy, se explicaran las propuestas de los principales motores gráficos del mercado y se compararan teniendo en cuenta las necesidades específicas de este proyecto. Se elegirá uno para el proyecto, y se documentará el proceso de adaptación y aprendizaje que se ha llevado a cabo durante la producción del videojuego.

1 Un *framework* de *software* es una estructura que actúa como plantilla o punto de partida para la organización o el desarrollo de software [2] .

2 En el desarrollo de *software*, que algo sea de bajo nivel indica que tiene menos capas de abstracción respecto al *hardware*.

2.1.2 Historia

Si estamos repasando la historia de los motores gráficos en este apartado es porque vale la pena entender de donde vienen y qué pasos se han dado en la industria del desarrollo de software de videojuegos para que en la actualidad los motores gráficos sean un concepto tan presente en el desarrollo de videojuegos, tanto en la escena independiente como en la industria *AAA*[3].

En los albores de la industria no existía el concepto de motor gráfico, ya que los juegos se programaban como un único paquete del que no se esperaba poder reutilizar nada. Esto se debe a que las primeras generaciones de consolas (estamos hablando de finales de los 70 hasta mediados de los 80) tenían enormes limitaciones en cuanto a memoria, lo cual provocaba que los programadores tuvieran que ser extremadamente cuidadosos con su uso. Esto limitaba bastante las posibilidades de abstracción y reutilización, ya que había que construir cada juego teniendo muy en cuenta las necesidades gráficas de este y como aprovechar al máximo el *hardware* de la consola para poder llevarlas a cabo. Con la capacidad de la tecnológica moderna, esa necesidad se ha ido diluyendo, siendo realmente importante solo en los títulos más demandantes.

Figura 1: La Atari 2600 (1977) tenia 128 bytes de RAM y sus cartuchos podían almacenar hasta 4 KB [4] Fuente: https://en.wikipedia.org/wiki/Atari_2600.

Además, otro factor muy importante para que no se planteara la creación de los motores durante esta época es que la parte más importante del mercado de los videojuegos no era la doméstica (consolas y ordenadores) sino los árcades. Durante los primeros años de los árcades de videojuegos, de principios de los 70 hasta inicios de los 80, el *hardware* de estas máquinas avanzaba a gran velocidad, provocando así que las diferentes generaciones de árcades salieran muy cercanas entre sí. Esto desincentivaba el escribir código con el objetivo de reutilizarse, ya que al tener que desatollar juegos para las nuevas generaciones se tenían que programar y diseñar de manera completamente diferente para poder aprovechar la tecnología más avanzado de estas nuevas máquinas .

3 Un videojuego AAA es una clasificación informal utilizada para los videojuegos producidos y distribuidos por una distribuidora importante o editor importante, típicamente teniendo marketing y desarrollo de alto presupuesto [3]

A mediados de los 80 empezaron a aparecer las primeras piezas de software que podríamos catalogar como motores gráficos, aunque el término (*Game Engines*) no se acuñaría hasta los 90. Estos motores serían todos desarrollados internamente por las empresas de videojuegos para crear sus proyectos, aunque más tarde se popularizaría el trabajar en estos motores para poder licenciarlos para otras empresas.

Uno de los primeros ejemplos del que tenemos constancia es el motor utilizado en el juego de carreras de NES, *Excitebike* (1984), creado por el equipo de Shigeru Miyamoto en Nintendo. Este motor fue creado para desarrollar juegos de *scroll* lateral, llamados así porque la cámara se mueve suavemente por la pantalla permitiendo un avance continuado en vertical o en horizontal. Este motor fue utilizado más adelante en el conocido por todo el mundo *Super Mario Bros* (1985), aprovechando las capacidades de este motor para crear niveles en los que se avanza de izquierda a derecha mientras la cámara sigue a Mario [5] .

Otro ejemplo anterior al acuñamiento del término, pero muy importante durante los primeros años de la industria fue SCUMM, creado en 1987 por *Lucasfilm Games* para la *Commodore 64*, y que posteriormente fue publicado en todas las plataformas del momento. Este motor fue creado para ayudar con el desarrollo de *Maniac Mansion*, una de las primeras aventuras gráficas de la historia, y que se utilizaría en los siguientes juegos de la empresa durante más de diez años. El motor está diseñado en torno a un sistema de verbos que ofrecen al jugador diferentes acciones para interactuar con el escenario y los personajes [6] .

Figura 2: Maniac Mansion (1987). La franja en la parte inferior de la pantalla muestra el icónico sistema de verbos de SCUMM. Fuente: https://en.wikipedia.org/wiki/Maniac_Mansion

El término motor gráfico se empezaría a popularizar con la llegada de *Doom* (1993), un juego de disparos en primera persona desarrollado por *id Software*. Conocido internamente como el *id Tech 1*, este motor traía la novedad de poder proyectar planos en dos dimensiones para simular un entorno de tres dimensiones. Pero importante para este repaso no son las capacidades del motor, sino el uso que se le dio. Fue un motor inmensamente popular en su momento, ya que *id Software* comenzó a licenciar el motor para permitir que otras empresas lo utilizaran para sus juegos [7] , y en 1997 el código del motor fue liberado bajo la licencia *GNU General Public License*, permitiendo a cualquier persona o empresa usarlo para desarrollar sus videojuegos. Además, este motor es de gran importancia histórica no solo por su uso sino porque fue la base para los siguientes motores de la compañía, como el *id Tech 2,* también conocido como el *Quake II Engine*, ya que fue el motor usado para desarrollar este

juego. Este motor sé utilitaria no solo para desarrollar el siguiente motor de la compañía, el *id Tech 3*, sino que se le cederían los derechos a *Valve* para utilizarlo de base para crear el motor *GoldSrc*, que se usaría para crear *Half-Life* y sería la base de los siguientes motores de la compañía.

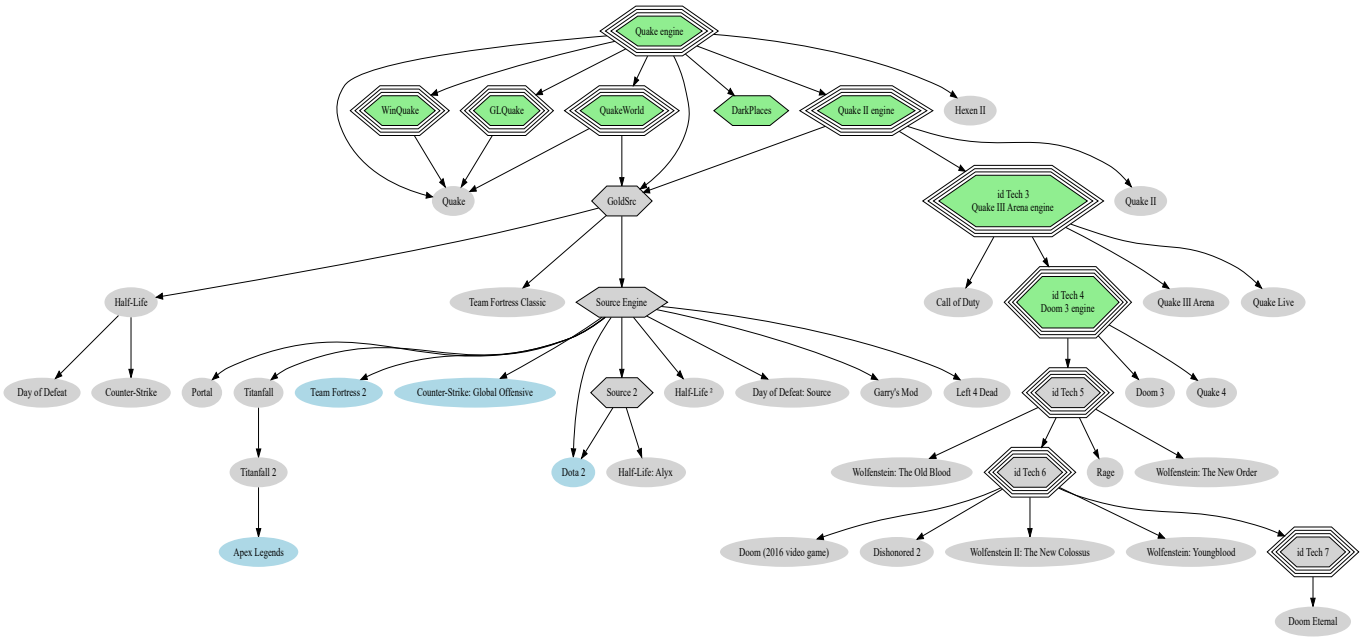

Figura 3: Esquema que muestra los diferentes motores y juegos que se desarrollaron en base al Quake Engine
Fuente: https://en.wikipedia.org/wiki/Game_engine

Además de la cantidad de juegos que fueron desarrollados con estos motores, también son muy importante porque fueron los precursores del modelo que se popularizaría años después en la industria. A partir del *id Tech 3*, se empezó a tratar por separado el desarrollo del juego y del motor, ya que había un énfasis mayor en poder licenciar el motor a otras empresas, que se beneficiaban enormemente de las comodidades que estos motores podían ofrecerles sin tener que invertir el tiempo y el I+D necesarios para desarrollar uno internamente.

Desde que se asentó este modelo, han salido muchos motores gráficos, tanto pensados para grandes empresas como para entusiastas y pequeños grupos de desarrolladores. En la industria AAA, los motores más populares a lo largo de los años han sido:

- *Unreal Engine*, creado por *Epic Games* en 1998 para *Unreal Tournament*. Ha tenido muchas iteraciones a lo largo de los años, llegando al actual *Unreal Engine 5*. Utiliza el lenguaje de programación *C++* para el *scripting*, aunque las últimas versiones le han dado un gran énfasis al sistema de programación visual basado en nodos llamado *blueprints*. Ha sido ampliamente usado en juegos de todos los géneros, aunque está pensado para trabajar en entornos de tres dimensiones.

- *CryEngine*, creado por *Crytek* en 2002. Actualmente, la empresa está trabajando en *CryEngine 6*. Utiliza los lenguajes de programación *C++* y *C#* para el *scripting.* Este motor se hizo muy conocido por sus avanzadas tecnologías de renderizado foto-

realista. Está pensado para desarrollar principalmente juegos de disparos en primera persona.

- *Source*, creado por *Valve* en 2004. Es el sucesor del *GoldSrc*, y fue diseñado para crear *Half-Life 2*. Utiliza el lenguaje de programación *Lua para* el *scripting.* Aunque el motor sea conocido principalmente por los juegos de su empresa creadora, *Valve*, muchos estudios pequeños e independientes han utilizado este motor tanto para crear juegos nuevos como para modificar juegos existentes. Este motor es conocido por sus avanzadas técnicas de iluminación y por el uso del motor de físicas *Havok* para tener las simulaciones de físicas más avanzadas vistas hasta el momento.

Si quitamos la vista de la industria AAA y nos centramos en el mercado independiente, en el que priman juegos creados por un pequeño equipo de desarrolladores o incluso por una sola persona, veremos que el panorama es bastante diferente. Los motores más populares en este ámbito han sido los siguientes:

- *RPG Maker*, creado en 1992 por *ASCII*, es un motor pensado para hacer únicamente juegos del género *JRPG*, al estilo de los primeros *Dragon Quest*. Utiliza los lenguajes *Ruby* y *Javascript* para el scripting. Viene con un sistema de combate, equipamiento y grupo implementado por defecto, junto con varios recursos gráficos y sonoros para que puedas empezar a desarrollar el juego con la mayor rapidez posible. Está pensado para poder crear juegos incluso si no se sabe programar, y ha sido muy influyente en la creación de videojuegos independientes en los últimos veinte años, aunque su uso cada vez es menor.

- *GameMaker*, creado en 1999 por *Mark Overmars*. Es un motor de uso general, aunque limitado a juegos en dos dimensiones, y utiliza el *Game Maker Language* (*GML*) como lenguaje de *scripting*. El principal atractiva de este motor es que combina el *GML* con un sistema de programación visual bastante avanzado que permite que gente sin conocimientos previos de programación pueda crear sistemas complejos. La última versión del motor se publicó en 2022, y durante los últimos años ha sido uno de los motores más utilizados para la publicación de juegos independientes.

- *Unity,* creado por *Unity Technologies* en 2005, es el indiscutible líder en cuanto a videojuegos independientes. Aunque también ha sido bastante utilizado por grandes empresas, este motor lleva muchos años siendo el más utilizado para el desarrollo independiente. Utiliza el lenguaje *C#* para el *scripting*, y está preparado para desarrollar juegos de todo tipo de géneros, tanto en tres como en dos dimensiones. Es muy versátil y contiene multitud de herramientas. En 2023 se publico Unity 6, la última versión hasta el momento.

- *Godot,* creado por *Juan Linietsky* en 2014. El lenguaje de scripting que utiliza es *GDScript*, un lenguaje propio creado específicamente para *Godot,* pero se pueden utilizar muchos más lenguajes gracias al sistema *GDExtension*, que permite exponer la API del motor a cualquier otro lenguaje. Este motor sigue la estela de *Unity*, con la intención de servir para desarrollar cualquier tipo de juego, con la diferencia de que

este es *FOSS*[4], bajo la licencia *MIT*[5]. Actualmente, se encuentra en su versión 4.3, publicada en septiembre de 2024.

Desde finales de los 90 y durante los siguientes veinticinco años, la industria de los videojuegos tendrá un equilibrio entre empresas que desarrollan sus motores internamente y empresas que compran una licencia para utilizar el motor de otra compañía. No es hasta los últimos años en los que se está empezando a notar un descenso en el número de empresas que deciden crear su propio motor mientras aumenta el uso de motores de tercero, siendo el más utilizado *Unreal Engine 4* y *5.*

4 Free and Open Source significa que el programa es gratuito y de código abierto.

5 La licencia *MIT (Massachusetts Institute of Technology) es un tipo de licencia permisiva que no pone restricciones al usuario con lo que puede y lo que no puede hacer con el producto.*

2.1.3 Investigación

En este apartado se repasarán todos los motores que se han tenido en cuenta a la hora de desarrollar este trabajo y se listarán todas las características que puedan resultar importantes para el desarrollo del videojuego.

2.1.3.1 Unreal Engine

Aunque este motor se haya mencionado en la lista de los AAA en el apartado anterior, tiene opciones de licencia gratuita y desde la versión 4 su uso en la industria independiente se ha visto en aumento.

Está disponible para Windows, Linux y macOS. Puede crear juegos para estas tres plataformas, móviles, consolas y realidad virtual. Es extremadamente potente a nivel gráfico, pero a cambio se necesita un ordenador con grandes capacidades gráficas y computacionales para poder utilizarlo sin problemas.

Su principal punto de venta es la fidelidad gráfica que se puede llegar a obtener utilizando este motor, gracias a sus avanzados sistemas de iluminación y técnicas de renderizado foto realista.

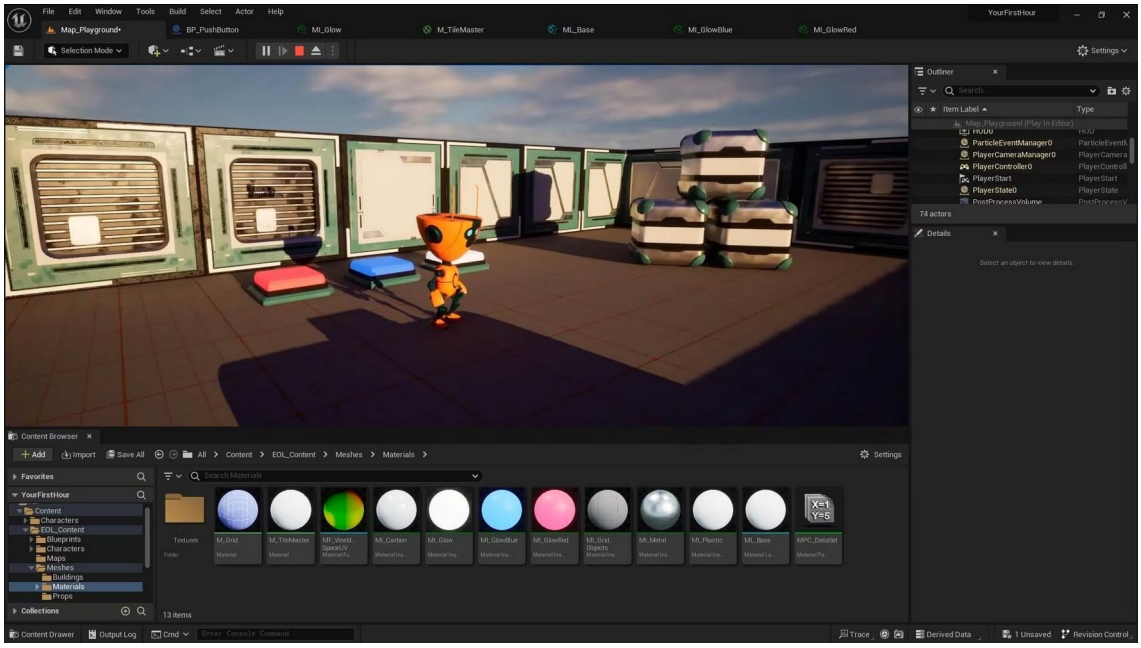

Figura 4: Interfaz de Unreal Engine 5. Fuente: https://www.unrealengine.com/en-US/unreal-engine-5

Como añadido de su última versión, incluye el sistema *Nanite*, que permite renderizar objetos en tres dimensiones sin tener que preocuparse por la cantidad de triángulos que los compongan, lo que ahorra tiempo al no tener que optimizar la carga poligonal de los modelos. Actualmente, este sistema está limitado a mallas estáticas que no se deformen, así que aún no puede ser utilizado para renderizar personajes.

Aunque este motor este claramente enfocado al foto-realismo, se puede utilizar para crear estéticas estilizadas que no busquen el realismo, e incluso se pueden hacer juegos en dos dimensiones, aunque muchas de las herramientas que trae no están especialmente preparadas para ello.

Otro de los puntos que atraen a muchos usuarios a este motor es el sistema de *blueprints*, un sistema de programación visual basado en nodos con el que puedes crear patrones de programación complejos conectando los diferentes nodos entre sí.

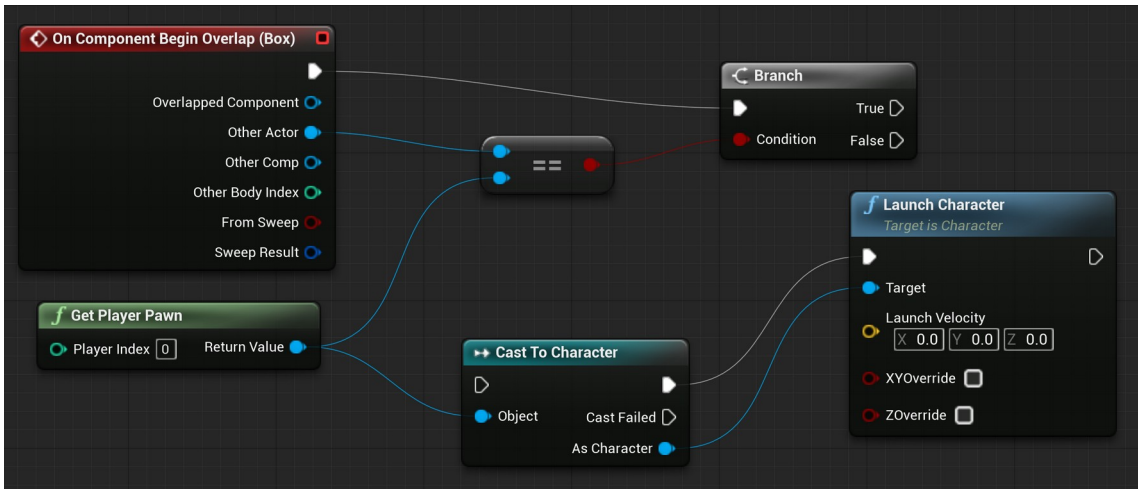

Figura 5: Pequeño ejemplo del sistema de *blueprints* de Unreal Engine. Fuente: https://www.unrealengine.com/en-US/unreal-engine-5

Este sistema es complementado por *C++*, un lenguaje de programación extremadamente popular y potente que permite tener un gran grado de control sobre el uso de la memoria, lo que puede llegar a ser muy importante en los juegos de gran escala para los que se usa este motor.

Además, cuenta con el *Unreal Engine Marketplace*, una tienda de recursos listos para ser utilizados dentro del motor con multitud de recursos muy útiles a la hora de desarrollar un videojuego. Tiene una sección de recursos gratuitos, y suelen poner algunos recursos de pago gratis durante tiempo limitado.

Para nuevos usuarios interesados en aprender a manejar el motor, hay una gran cantidad de recursos de aprendizaje en internet, tanto oficiales de *Epic Games* como hechos por la comunidad.

En cuanto a la licencia, funciona a través de regalías: si obtienes menos de un millón de dólares americanos de ingresos anuales con un producto creado con el motor, no tiene ningún coste. A partir del millón de dólares, *Epic Games* te cobra un 5% de los ingresos del juego. Esto no aplica a los centros educativos, para los que la licencia siempre es gratuita [9] .

Es un motor muy completo, potente y versátil, ampliamente utilizado por grandes y pequeñas empresas, y una de las piezas clave de la industria de los videojuegos en estos momentos. Tiene un foco bastante claro en las grandes producciones y los gráficos realistas.

2.1.3.2 Unity

Unity tiene el caso contrario al motor que acabamos de ver; es usado ocasionalmente en la industria AAA, pero su fuerza está en el mercado independiente. Desde hace aproximadamente diez años es el motor gráfico con la mayor cuota de mercado, estando aproximadamente el 15% de los juegos de Steam creados con él [8] . Está disponible en

Windows, Linux y macOS, y puede usarse para crear juegos para estas tres plataformas, consolas, móviles, realidad virtual y navegadores web.

Tiene soporte para juegos de dos y tres dimensiones y incluye tres *pipelines* gráficas[6] cada una capaz de renderizar gráficos con una fidelidad visual mayor que la anterior, pensadas para que el motor pueda adaptarse a todas las plataformas. Esto le permite ser muy versátil y poder utilizarse para todo tipo de juegos, desde pequeños juegos móviles hasta experiencias hiper-realistas en consola. Es gracias a esto que es con mucha diferencia el motor más utilizado en juegos de móvil, acumulando aproximadamente el 50% de la cuota de mercado [11] .

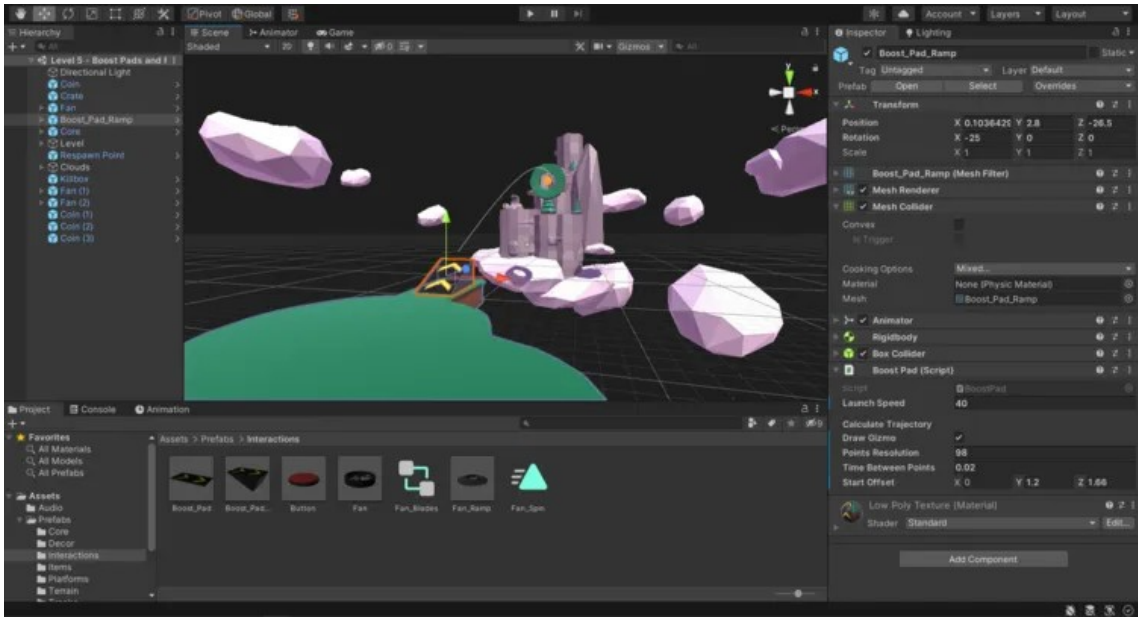

Figura 6: Interfaz de Unity 2023. Fuente: https://unity.com/es

En general, está considerado como un motor con una curva de aprendizaje suave que permite a los recién iniciados en la materia utilizarlo sin demasiados problemas. Aunque sea fácil iniciarse con él, es un motor muy complejo y difícil de dominar al completo.

Utiliza el lenguaje de programación *C#*, un lenguaje orientado a objetos que, al igual que el propio motor, este relativamente fácil de empezar a utilizar, tiene una sintaxis parecida a *Java* y como este, tiene un «recolector de basura», que se dedica a manejar la memoria en segundo plano. Esto hace que no tengas que tener en cuenta como administrar la memoria, al contrario que en *C++*, pero se traduce en un esfuerzo extra a nivel de computación ya que el recolector tiene que estar procesándose de fondo todo el rato.

A partir de la versión *Unity 2021*, se introdujo una herramienta de *scripting* visual llamada *Unity Visual Scripting* para complementar este lenguaje, pero parece ser demasiado simple como para utilizarlo para programar sistemas complejos, y la mayoría de usuarios lo utilizan únicamente en la fase de prototipado. Además, no se ha vuelto a actualizar desde la versión 2021.3.

6 La *pipeline* gráfica realiza una serie de operaciones para renderizar en la pantalla los contenidos de una escena [10] .

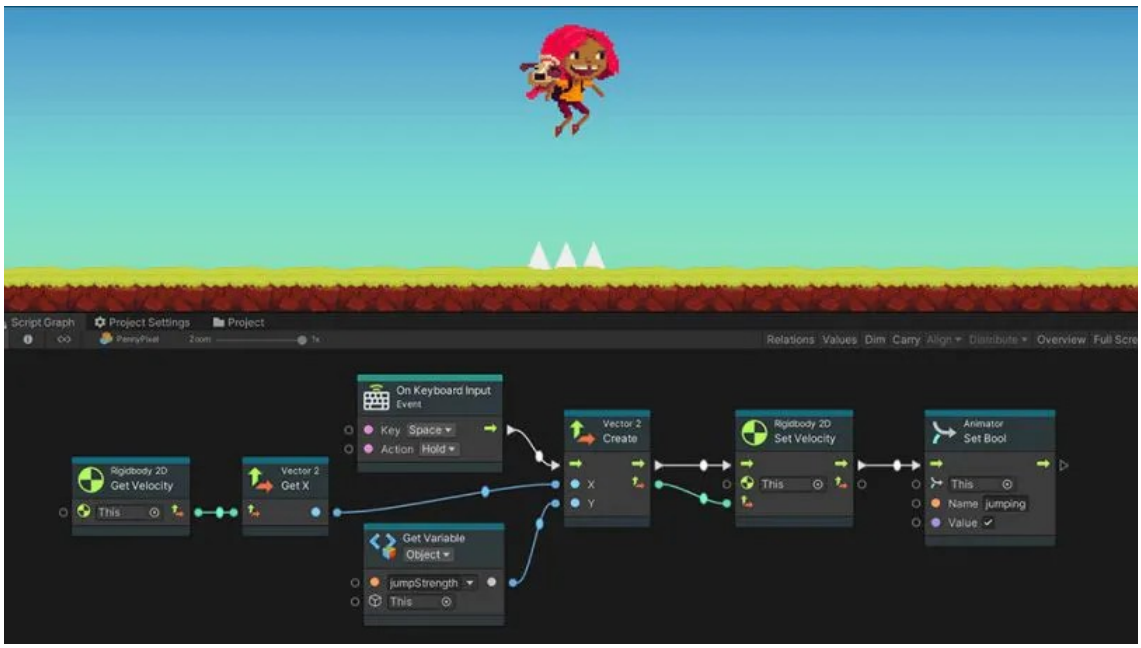

Figura 7: Un *script* de movimiento en Unity Visual Scripting. Fuente: https://unity.com/es

Unity cuenta con la *Unity Asset Store* un mercado virtual en el que hay una enorme variedad de recursos tanto gratuitos como de pago para facilitarte la experiencia de crear un videojuego con este motor. Los productos en esta tienda están muy bien organizados, con diferentes métricas y filtros para ayudarte a descubrir los que mejor se adapten a tu proyecto. Además, existe una gran comunidad de gente que se dedica a probar y revisar estos productos para identificar los más útiles.

Sobre todo, esa es la mayor virtud que tiene este motor, la enorme comunidad de usuarios y creadores que lo utilizan. Es muy sencillo encontrar información en internet sobre cualquier cosa relacionada con el motor, desde como utilizar ciertas funciones a diferentes errores que te puedes encontrar mientras lo utilizas. En cuanto a tutoriales y materia de aprendizaje, es con mucha diferencia el motor que más tiene; puedes encontrar tanto en foros, Y*ouTube* y páginas de cursos instrucciones paso a paso de como hacer casi cualquier cosa. Está tan asentado que es el motor que se enseña normalmente en los centros educativos que imparten materia sobre desarrollo de videojuegos, como por ejemplo en la Salle.

En cuanto a la licencia, el tema es un poco confuso ahora mismo. Durante el desarrollo de este trabajo, la empresa que desarrolla el motor, *Unity Technologies*, ha hecho varios cambios en la política de licenciado del motor que han resultado ser muy polémicos. Hasta ahora, la licencia al programa era una suscripción anual que tenías que pagar si tu juego superaba los 200.000 dólares americanos de beneficio anual. En el caso contrario, podías usar el programa con la licencia gratuita. Esto se sigue manteniendo, pero en septiembre de 2023, se publicaron las nuevas condiciones para las licencias, que debido a la reacción de la comunidad han pasado por varios cambios y transformaciones. Para cuando se entregue este documento, las condiciones llevarán varios meses sin haber cambiado, así que es bastante seguro asumir que se quedaran así durante algún tiempo. Ahora mismo, están condiciones indican que, a partir de que tu juego haya generado un millón de *engagements* (más información más adelante sobre que quiere decir esto) y haya obtenido un millón de dólares de ingresos totales en los últimos 12 meses, tendrás que elegir entre pagar un 2,5% de los ingresos anuales del juego, o

pagar la *Runtime Fee*, que es como llaman a su tarifa basada en cuanta gente juega a tu juego cada mes [12] .

Esto se determina basado en los *engagements*, uno de los conceptos más polémicos que trajeron estos cambios. Según su propia página web, estos se definen como «momento en que un usuario final distinto adquiere, descarga o se involucra con éxito y legítimamente con un juego impulsado por *Unity Runtime*, por primera vez en un canal de distribución. [12] ». Estos datos tiene que reportarlos la empresa creadora del juego.

Estos cambios solo se aplicarán a partir de la siguiente versión del programa, *Unity 6*, aunque el plan inicial era que entraran en vigor de manera retroactiva para todas las versiones anteriores del programa, otra medida muy polémica que ha sido revertida.

Unity ha sido durante mucho tiempo, y sigue siendo hoy en día, el motor de referencia para los creadores independientes y la primera recomendación que recibe todo el mundo cuando quiere adentrarse en el mundo del desarrollo de videojuegos. Es un motor muy complejo y versátil que satisfará las necesidades de casi todo el mundo, independientemente de para que plataforma se esté desarrollando, pero el reciente fiasco con las licencias ha mermado enormemente la confianza de los usuarios en la empresa, y lo ha convertido en un motor menos atractivo para las empresas AAA.

2.1.3.3 GameMaker

Este motor tiene presencia exclusivamente en el ámbito independiente, pero eso no lo hace menos importante. *GameMaker* se presenta como un motor muy fácil de aprender, pensado para que cualquier persona, independientemente de su bagaje técnico sea capaz de utilizarlo.

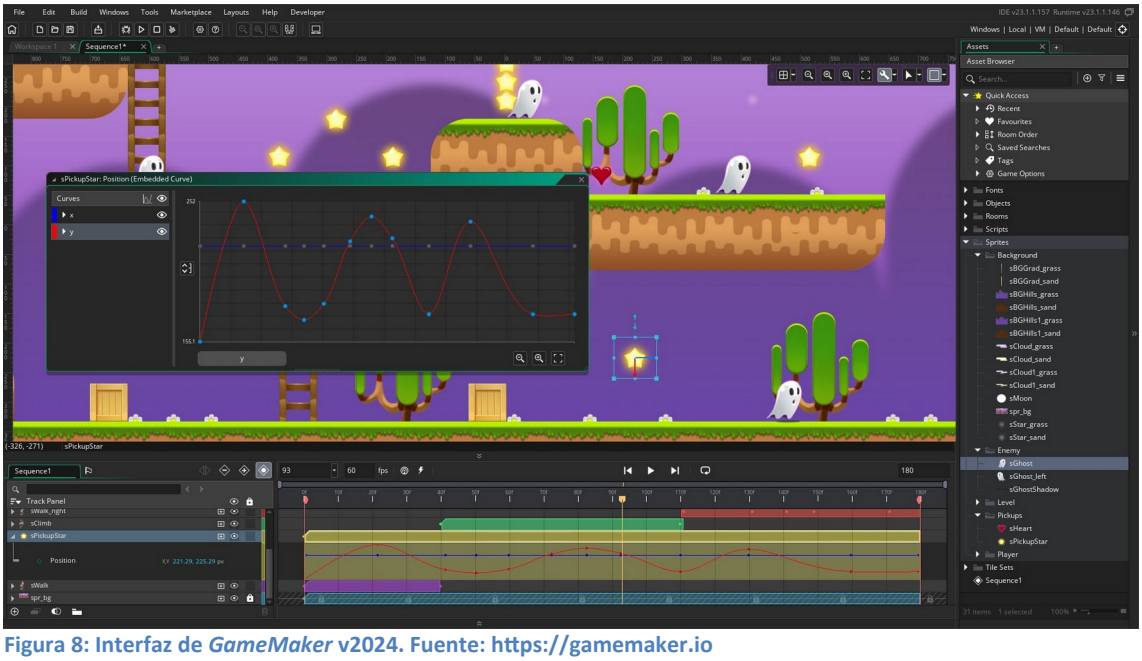

Figura 8: Interfaz de *GameMaker* v2024. Fuente: https://gamemaker.io

Está disponible para Windows, macOS y en cuanto a Linux, oficialmente solo tiene soporte para Ubuntu, pero se debería poder instalar en cualquier distribución. Puede usarse para crear juegos para estas tres plataformas, consolas, móviles, realidad virtual y navegadores webs, y está diseñado para crear juegos en dos dimensiones. Tiene soporte para 3D, pero es bastante limitado y no está pensado para que los usuarios novatos sean capaces de utilizarlo.

En este motor se trabaja principalmente con diferentes tipos de *sprites*[7], e incluye un sistema de huesos llamado *Spine* que permite animar figuras en dos dimensiones similar a como funcionan los esqueletos en la animación en tres dimensiones.

El motor incluye un IDE dentro del propio editor, para poder empezar a programar sin necesidad de programas externos, un editor de imágenes para poder crear y editar los *sprites* que se usan en el juego, y una herramienta para facilitar el diseño de niveles y la creación de mapas utilizando un sistema de casillas. Esto ayuda a que los usuarios que estén empezando en el desarrollo de videojuegos puedan realizar una gran cantidad de tareas con un solo programa y no tengan que aprender a utilizar varios programas al mismo tiempo.

En *GameMaker* se utiliza el lenguaje de programación GML, un lenguaje creado para este programa pensado para ser simple y fácil de aprender, pero muy potente si sabes como utilizarlo, con una sintaxis parecido a *JavaScript [14]* .

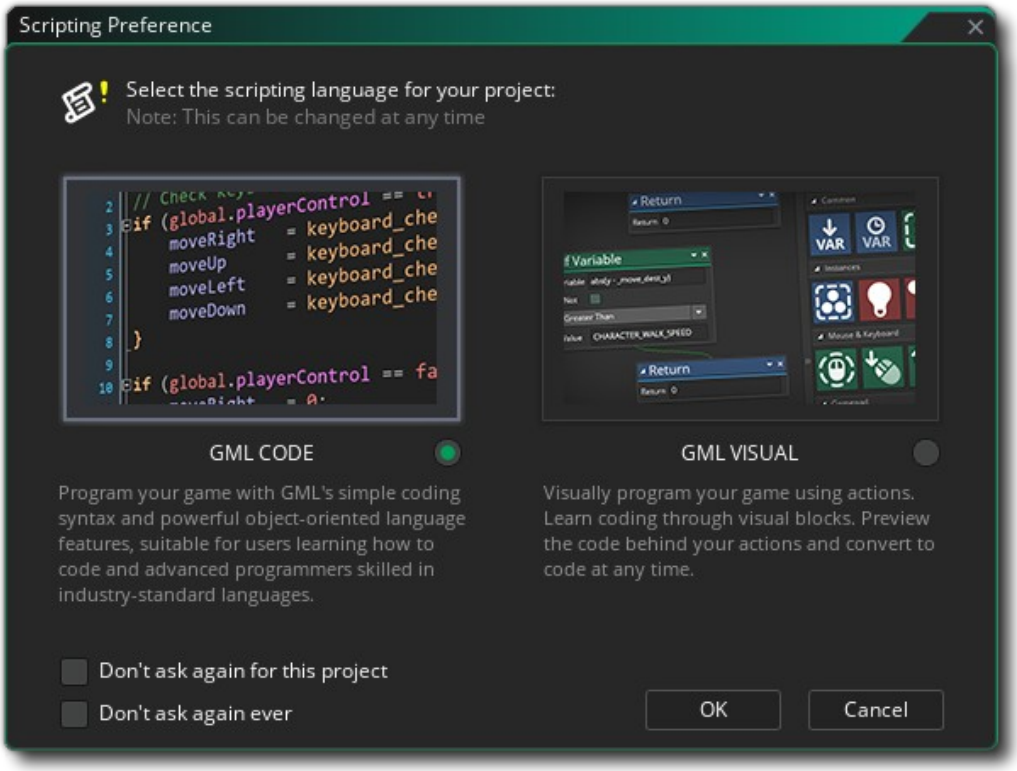

Figura 9: Al crear un *script GameMaker* te pregunta cual de los dos lenguajes prefieres utilizar. Fuente: https://gamemaker.io/en

Además, también incluye *G*ML Visual, un lenguaje de programación visual, no tan potente como su contraparte, pero muy útil para que los recién llegados al programa se acostumbren al lenguaje y al paradigma de programación que propone.

7 Los *sprites* (del inglés, "duendecillos") se trata de un tipo de mapa de bits dibujados en la pantalla de ordenador por *hardware* gráfico especializado. Típicamente, los *sprites* son usados en videojuegos para crear los gráficos [13] .

Este motor también cuenta con una tienda en la que obtener recursos para ayudarte con el desarrollo de tu juego, aunque no es tan extensa ni está tan bien planteada como las otras dos que hemos visto hasta ahora. Para aprender a utilizar este motor, hay una gran cantidad de información en internet, tanto en forma de videos como de artículos, explicando como se utiliza. La comunidad que usa este software es muy extensa, no es difícil resolver cualquier duda o problemas que puedas tener a base de buscar o preguntar en foros o páginas especializadas y es fácil encontrar cursos tanto gratuitos como de pago que enseñan todo lo que hay que saber sobre el motor.

En cuanto a la licencia, a partir de 2024 se cambió su funcionamiento: ahora el motor es gratuito para juegos no comerciales, excepto si quieres sacarlos en consolas, la licencia profesional ha pasado de ser una suscripción anual a un pago único de 99 dólares americanos, y para poder desarrollar consolas hay que seguir pagando la licencia *enterprise* de 80 dólares al año [15] .

En general, *GameMaker* es un motor perfecto para gente con poca o nula experiencia en el campo del desarrollo de software y también una muy buena opción para desarrolladores más experimentados que no quieran salirse de las dos dimensiones. Trae varias herramientas que pueden hacerte la vida más fácil como desarrollador, y ha demostrado que es capaz de crear juegos espectaculares con una muy baja barrera de entrada.

2.1.3.4 Godot

Godot es el más joven de todos los motores que se han mencionado aquí, y tiene una diferencia fundamental respecto al resto de motores: es **FOSS**. Es un motor generalista, pensado tanto para las dos y las tres dimensiones, y aunque hasta ahora no era muy conocido, ha empezado a ganar tracción en los últimos doce meses, impulsado en gran parte por la controversia con las licencias de *Unity*. El motor está disponible en Windows, Linux, macOS, Android, Web (se puede utilizar desde el navegador, sin necesidad de descargarlo), FreeBSD, NetBSD y OpenBSD [16] . Al ser tan joven en comparación con el resto de motores de esta lista, aún no hay demasiados juegos publicados que utilicen este motor.

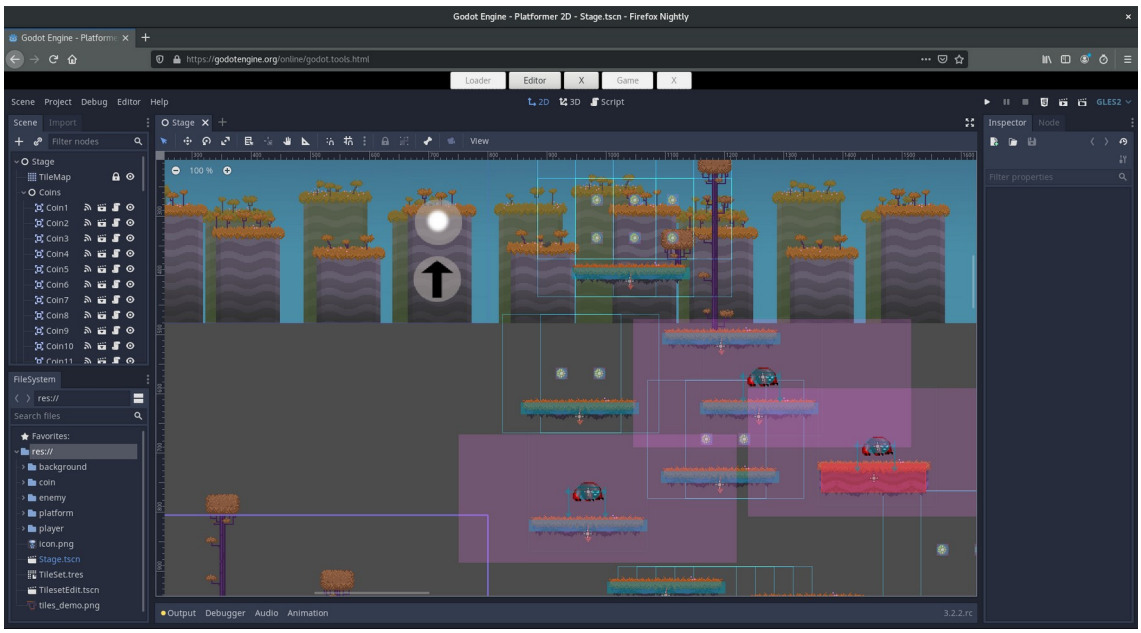

Figura 10: Godot 3 ejecutándose desde un navegador web. Fuente: https://godotengine.org/

Se puede utilizar para crear juegos para todas las plataformas anteriormente mencionadas y iOS. Actualmente, por limitaciones de la licencia *MIT* a la que está sujeto el software, no puede utilizarse para desarrollar juegos para consolas, ya que eso supondría tener que publicar el código del *SDK* estas. No obstante, se está trabajando en una solución para este problema.

La curva de aprendizaje de este programa es similar al resto de motores generalistas como *Unity* y *Unreal*, pero al estar más limitado en características, puede ser menos intimidantes pare los recién llegados. Hay que tener en cuenta que estos dos motores tienen varias características pensadas para la creación de efectos especiales y renderizado de animaciones para cine y otras industrias, mientras que *Godot* se centra únicamente en videojuegos.

El lenguaje de programación que se utiliza principalmente en *Godot* es *GDScript*, un lenguaje creado para este software, diseñado para integrarse completamente en el editor (el propio editor está programado en *GDScript*) y ser fácil de aprender y cómodo de utilizar. Tiene una sintaxis que recuerda a *Python*, y permite asignar tipos a los objetos de manera dinámica o estática a elección del usuario, aunque la documentación oficial recomiendo especificar siempre que sea posible los tipos de manera estática. Además, el motor incluye de manera oficial soporte para *C#,* que aunque no llega al mismo nivel de integración con el editor que tiene *GDScript*, funciona perfectamente y puede utilizarse para programar sin ningún tipo de limitación. La documentación oficial siempre incluye ejemplos en ambos lenguajes. Un añadido muy particular de este motor es el sistema *GDExtension* que permite exponer la *API* interna del motor (que está programado en *C++)* para permitir diseñar extensiones para poder programar en el motor utilizando cualquier lenguaje de programación. Actualmente, los creadores del motor han creado una extensión para programar en *C++*, que ellos mismos usan para añadir nuevas funcionalidades al motor, y existen muchas más creadas por la comunidad que permiten utilizar lenguajes como *Rust*, *Javascript*, *Lua, y más [17]* . El editor incluye un *IDE* para trabajar con *GDScript*.

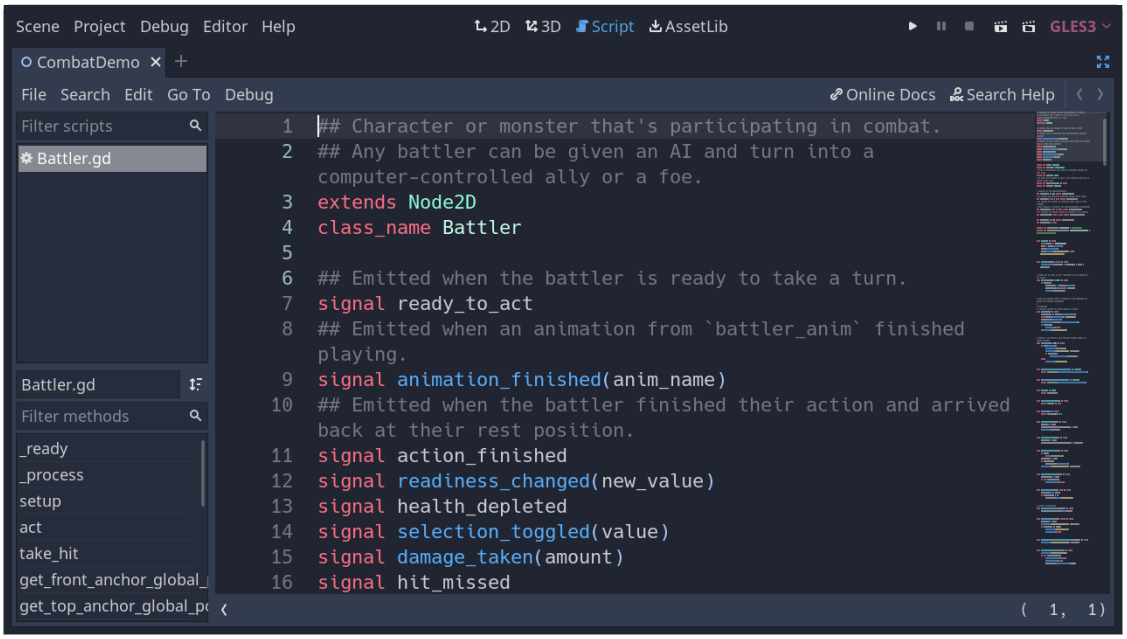

Figura 11: Interfaz de Godot 4 con el editor de código abierto. Fuente: https://godotengine.org/

En versiones anteriores del editor existía un lenguaje de programación visual llamada *VisualScript*, pero fue desechado con el paso a *Godot 4* y en estos momentos no parece que sea una prioridad para el equipo de desarrollo, aunque existen intentos comunitarios por intentar crear una nueva versión.

A nivel de funcionalidad, las capacidades 2D del motor son excelentes, incluyendo varias herramientas para posicionar y trabajar con *sprites* de manera cómoda, y superando en rendimiento al resto de motores. En cuanto a las tres dimensiones, se queda más corto, con una *pipeline* de renderizado muy potente, pero que aún no es capaz de igualar a *Unity* en calidad ni rendimiento. Al igual que este último, tiene tres *pipelines* de renderizado para adaptarse a distintas plataformas, y actualmente se están haciendo muchos avances en mejorar la más potente de ellas, *Forward+.* En general, el desarrollo del motor avanza muy rápidamente, con nuevas versiones casi cada mes.

Aunque se está trabajando en ello, aún no existe una tienda de recursos como en el resto de motores aquí mencionados. Ahora mismo solo está la *AssetLib*, que incluye recursos gratuitos hechos por la comunidad, y que tiene recursos muy útiles, pero es difícil de navegar, le faltan muchas opciones de filtrado para encontrar lo que buscas y no tiene ninguna manera de guardarte artículos ni dejar ningún tipo de *feedback.*

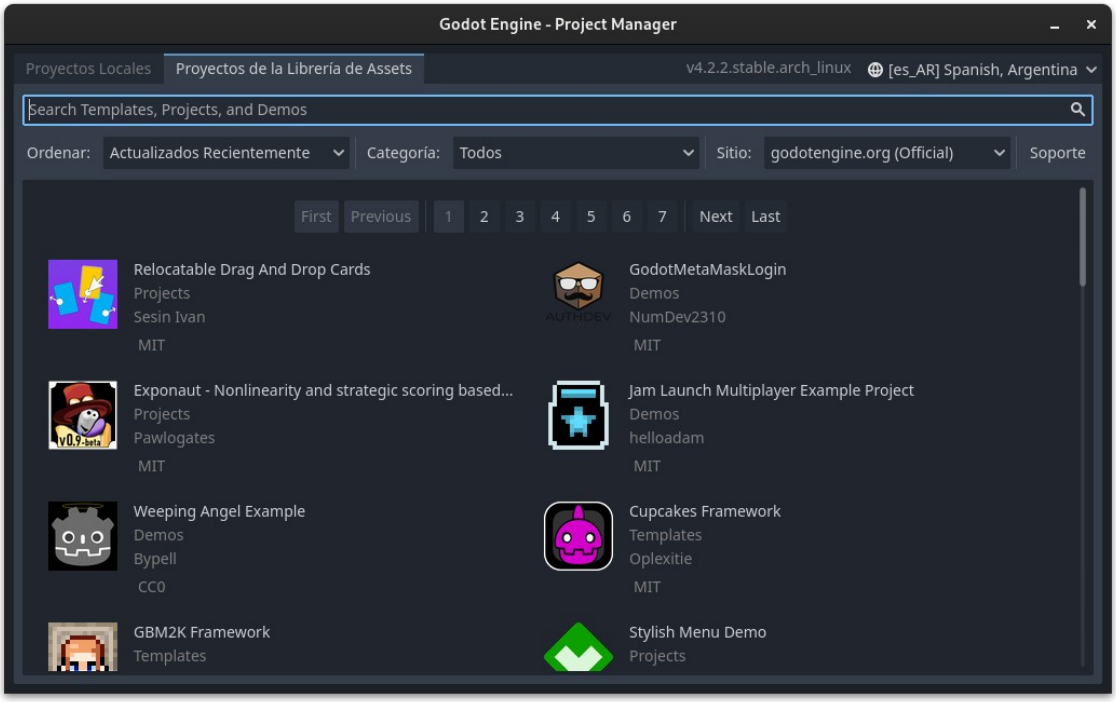

Figura 12: Librería de *assets* de Godot

A la hora de buscar información y ayuda con este motor, se nota que no lleva gozando de popularidad demasiados años. Al buscar cursos y tutoriales, te encuentras muchos, pero no tantos que te enseñen a usar la última versión, *Godot 4*, que tiene muchas diferencias significativas respecto a la versión anterior. Además, aunque la comunidad es muy entregada, es bastante común buscar soluciones a problemas muy específicos, y encontrarte que eres la primera persona en reportar tener ese error, y en general que en comparación con el resto de motores la información que existe en internet sobre este es bastante limitada. Por suerte, la documentación oficial es excelente, aunque ciertas partes del motor que no están completamente documentadas.

En cuanto a la licencia, aquí entra en juego lo primero que se ha mencionado sobre este motor, es gratuito y de código abierto. Puedes entrar a su repositorio de *github* y ver el código en cualquier momento, puedes utilizar una versión modificada por ti mismo o por otra persona, puedes hacer una *pull request* para arreglar errores o hacer mejoras, y en general, nadie puede decirte que puedes o que no puedes hacer con él, todo sin ningún tipo de coste asociado. Hay algunos límites, como que por ejemplo no puedes decir que tú has creado el motor, o que si construyes cualquier cosa utilizando este motor como base, por ejemplo creando una versión modificada del motor, tendrás que publicarla con la misma licencia que la del motor original.

En comparación con sus congéneres, este motor aún no esta completamente maduro, pero ha demostrado tener un enorme potencial y estar avanzando a un ritmo muy acelerado. Al ser de código abierto, que haya una creciente comunidad volcada en el se traduce en más gente trabajando en el motor. Para la escena independiente, sus capacidades gráficas son más que suficientes para la mayoría de juegos, aunque es incapaz de acercarse al estándar gráfico de la industria AAA. Es bastante cómodo de usar tanto para gente que está empezando como para

veteranos, y si sigue avanzando a este ritmo se puede convertir en un motor a la altura de los mejores.

2.1.4 Selección

Después de revisar los motores que podríamos considerar los más importantes en la industria independiente en estos momentos (sin tener en cuenta plataformas de creación que te limitan a la hora de publicar, como *Roblox*, *Fortnite* o *Core*) podemos decidir cuál utilizaremos para desarrollar este trabajo.

Hay que empezar definiendo las características que tendrá el videojuego que queremos crear. En este momento del trabajo aún no se han definido demasiadas características sobre este, pero sí que hay algunos aspectos que están decididos:

- Será un juego para móvil, específicamente para Android
- La escala del juego será pequeña, ya que será desarrollado por una sola persona en tiempo limitado.
- Tendrá un estilo gráfico estilizado y de poca fidelidad.
- Tiene que ser simple a nivel mecánico, idealmente se debería poder jugar con una sola mano.
- Debe poder funcionar en *hardware* humilde, ya que ni el móvil ni el ordenador que se usaran para el desarrollo son especialmente potentes.
- Puede ser tanto en tres como en dos dimensiones.

Con todos estos datos sobre la mesa, podemos hacer una decisión informada sobre qué motor utilizar.

Lo primero a tener en cuenta es la plataforma para la que queremos desarrollar el juego; todos los motores de la lista tienen capacidad para exportar juegos para Android, pero hay uno de ellos que flaquea ligeramente en cuanto al soporte móvil, y ese es *Unreal*. Eso no significa que él sea un mal motor para móviles, hay muchos juegos móviles creados con este motor y funcionan perfectamente, pero se nota que no es una de las prioridades del motor. Para empezar, *Unreal* hace que los juegos ocupen más que con otros motores, lo que puede ser un problema en el espacio móvil, y las herramientas de optimización que incluye no están tan bien planteadas para móvil como para el resto de plataformas. Si quieres hacer un juego que aproveche toda la potencia de los móviles modernos puede ser muy útil, pero para experiencias más reducidas este motor no parece ser la primera elección de nadie. Además, este motor podría tener problemas ejecutándose en el ordenador que va a ser usado para el desarrollo del juego. Por todo esto, el primer motor que se descarta en esta lista es *Unreal.*

El siguiente punto que puede ayudar a hacer una selección es el número de dimensiones a ser representadas. En este punto el juego no está limitado al 2D o al 3D, y se han planteado ideas que utilizan ambos, por lo que *GameMaker* sería demasiado limitante para el desarrollo. Aunque tenga cierto soporte para tres dimensiones, este es muy básico y muy poco práctico si lo comparamos con el resto de integrantes de la lista. Lo mismo se puede decir del soporte 2D de *Unreal.*

Esto nos deja con solo dos opciones: *Unity* y *Godot*. La elección entre ellos no es fácil, porque al nivel de uso que se le va a dar en este proyecto, ambos están muy parejos en cuanto a posibilidades. No se busca sacar partido de las *pipelines* gráficas más avanzadas de *Unity* y aunque el rendimiento en 3D de *Godot* está un poco por detrás, no se va a desarrollar un juego que vaya a poner al límite las capacidades de renderizado de ninguno de los dos motores. Ambos son idóneos para el tipo de juego que se desarrollara en este proyecto, así que llegados a este punto la elección queda a preferencias personales.

Personalmente, tengo cierta experiencia con *Unity*, y he encontrado tanto cosas que me gustan mucho como cosas que detesto del motor. Por otro lado, *Godot* es más nuevo y prometedor, y creo que este trabajo es una ocasión perfecta para probarlo. Esto hará de todo el proceso una experiencia más enriquecedora, ya que estaré aprendiendo un *software* nuevo que nunca hemos utilizado en la universidad. Como añadido, la versión de Linux de *Godot* funciona mucho mejor que la versión de *Unity*, y como este trabajo se realizara enteramente en *Linux*, hará el proceso más cómodo. Con todos estos datos, está decidido que el proyecto será desarrollado en ***Godot***.

2.1.5 Aprendizaje

Uno de los puntos en los que *Godot* palidece frente al resto de motores es la cantidad de material de aprendizaje que puedes encontrar, así que este proceso se puede hacer más complejo que con el resto de motores de los que hemos hablado. Esto no significa que no haya recursos, ni que no sean de calidad, como nos muestra la documentación oficial.

Figura 13: Extracto de la documentación oficial de la ultima versión no estable de *Godot* en español. Fuente: https://docs.godotengine.org/en/latest/index.html

Esta es el primer punto de ruta para aprender a utilizar el motor, está organizado para guiar a los recién llegados, empezando por las partes más fácilmente digeribles y aumentando en complejidad poco a poco. Incluye ejemplos de código de todo lo que explica, tutoriales y proyectos de ejemplo para ayudarte a entender los conceptos más importantes. La sección de «Primeros Pasos» es una parada obligatoria para ayudarte a entender como funciona el motor, y el manual incluye toda la información que hace falta conocer sobre las funciones. Está muy bien organizado e indexado para poder encontrar las secciones tanto desde la propia página como desde buscadores externos. Para este proyecto, esta ha sido la principal fuente de información y aprendizaje. Antes de ponerme a utilizar el motor por mi cuenta, seguí todos los tutoriales y leí la documentación de todas las funciones que creí que necesitaría usar, y cada vez que necesitaba información de como hacer algo accedía a esta página.

Aunque la documentación oficial sea excelente, a veces también hace falta información extra que no se encuentra allí o que es más fácil de ser consumida en otros medios, como por ejemplo en videos. Hay una comunidad bastante grande de creadores que suben videos explicando como utilizar el programa, o enseñando pequeños trucos y consejos para utilizarlo de manera más eficiente. En mi caso el video *«The ultimate introduction to Godot 4»*, del canal de *YouTube «Clear Code»*, un tutorial de 11 horas y media que explora todas las partes del motor, me fue inmensamente útil para familiarizarme con la manera de trabajar de este motor. También consulte los videos de *GDQuest*, un canal especializado en tutoriales de *Godot* que tiene una enorme cantidad de materia de aprendizaje. En general vi muchos videos que recopilan detalles y consejos que no son obvios para una persona recién llegada al motor, pero que son inmensamente útiles para ahorrar tiempo o para utilizar ciertas herramientas a todo su potencial.

Por supuesto, lo más útil para aprender como funciona el motor ha sido utilizarlo. Durante el desarrollo de los primeros prototipos me pude familiarizar mucho con el funcionamiento del motor y con los diferentes problemas que pueden surgir cuando lo utilizas, qué cosas se puede o no hacer, como organizar el proyecto para que pueda escalar sin problemas, qué paradigmas de código pueden hacerte la vida más fácil y largo plazo, y en general como hacer las cosas a la manera de *Godot.*

Uno de los conceptos que más únicos de *Godot* es que funciona con un sistema de «nodos» llamados escenas. En *Godot* absolutamente todo es una escena, y estas se organizan jerárquicamente en un sistema de árbol.

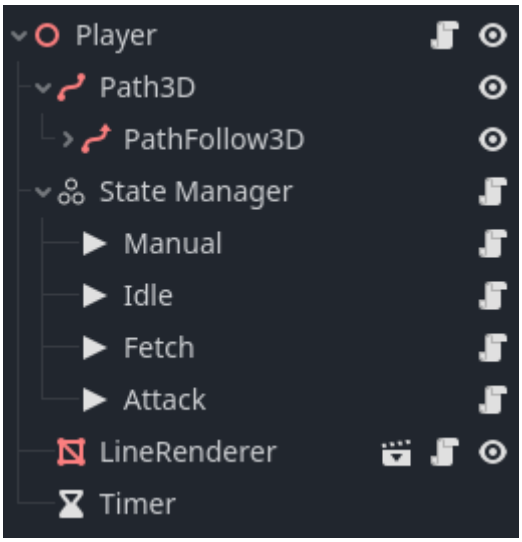

Figura 14: Ejemplo de un árbol de escenas en *Godot*

Estos pueden heredar características de otros nodos utilizando la herencia típica de los lenguajes orientados a objetos, y por como se organizan invitan a utilizar paradigmas de programación orientados a la composición, ya que se puede tratar cada uno de estos nodos como pequeñas piezas reutilizables que añaden funcionalidades al nodo padre. Este enfoque es un tanto diferentes al del resto de motores que se han comentado aquí, y puede ser uno de los principales puntos de fricción para los usuarios que vengan de estos.

2.2 Introducción a Godot

Para no tener que pararse más tarde a explicar como funciona *Godot* y que las diferentes explicaciones fluyan correctamente, en este apartado se repasaran las diferentes partes y funcionalidades del motor que entraran en juego más adelante.

2.2.1 Interfaz

La interfaz de *Godot* tiene cinco partes muy bien diferenciadas. En la parte superior izquierda tenemos una sección con dos pestañas, la primera de ellas la de escena, donde se muestra una todos los nodos presentes en la escena, indicando con un icono su tipo y su visibilidad, además de si tienen algún *script* adherido o si emiten alguna señal. La segunda, importar, te permite seleccionar las diferentes opciones de importación cuando tienes seleccionado un archivo importable, como una imagen o un modelo en 3D. Debajo de estas se encuentra el Sistema de

archivos, que muestra todos los archivos y carpetas en el directorio del proyecto y te deja navegar entre ellas, eliminarlas, crear nuevas y abrirlas.

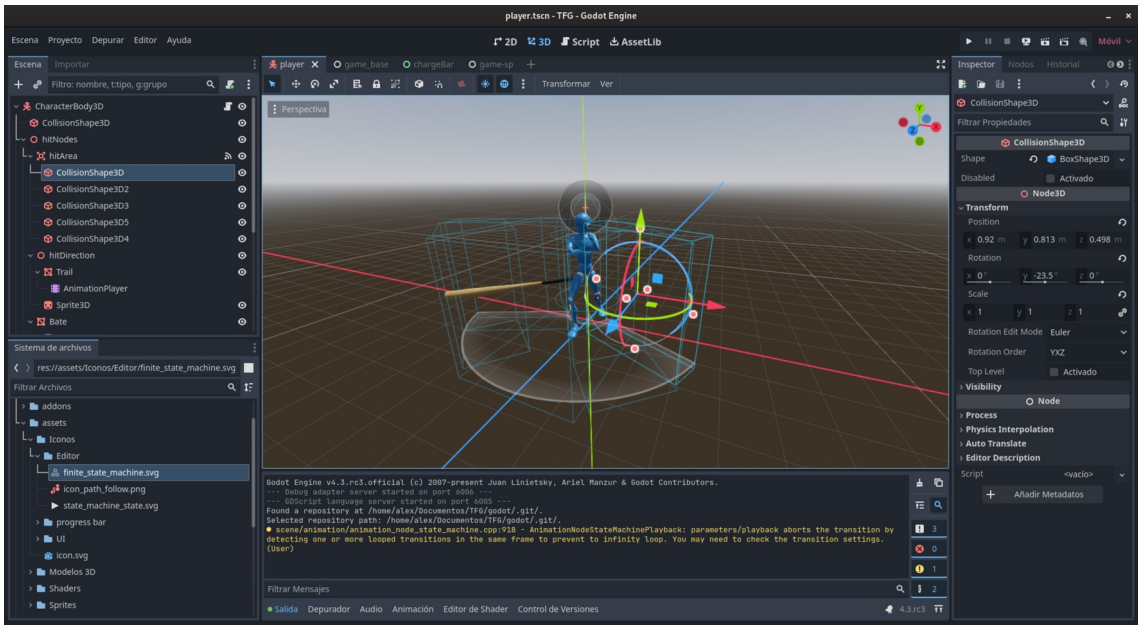

Figura 15: Interfaz de *Godot*

A la derecha se encuentra otra sección con varias pestañas: la primera de ellas es el inspector, que permite ver y editar todas las variables que el nodo seleccionado haya expuesto al editor. La siguiente pestaña, nodos, permite ver y conectar todas las señales que el nodo seleccionado es capaz de emitir y ver a qué grupos pertenece el nodo. La última pestaña, historial, simplemente te muestra todas las acciones que has realizado desde que has abierto el editor y te permite volver a un estado anterior pulsándolas.

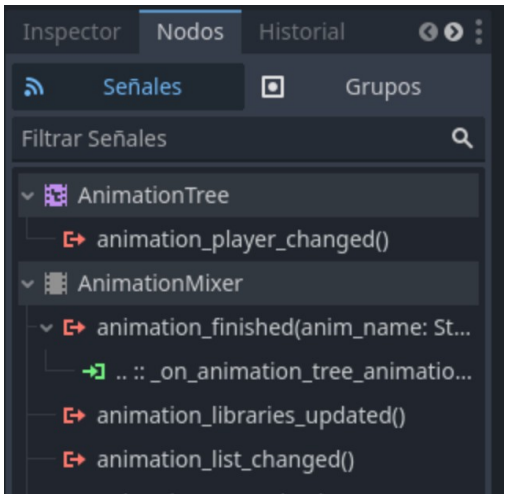

Figura 16: Pestaña nodos del editor de *Godot*

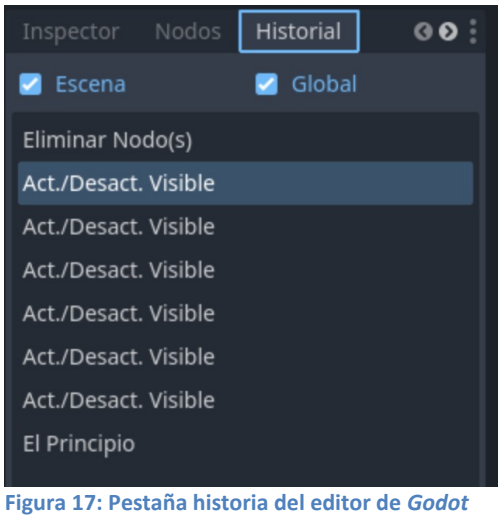

Figura 17: Pestaña historia del editor de *Godot*

En el centro de la pantalla encontramos la sección más importante, donde ocurre la mayor parte del desarrollo. Tiene cuatro pestañas, aunque las dos primeras cumplen el mismo propósito, renderizar los contenidos de la escena y permitirte interactuar con ellos. La primera te muestra todos los nodos en 2D y la segunda los nodos en 3D, pudiendo moverlos, rotarlos y escalarlos. La tercera es un editor de código que te permite programar los *scripts* sin tener que salir del editor, y tiene todas las funciones de un *IDE* moderno como un servidor de lenguaje,

autocompletado, salto a la definición, te permite ver tanto la documentación oficial como tu documentación propia sobre los diferentes nodos y funciones y en general está muy bien integrado con el resto del motor. La última pestaña se ha mencionado en los apartados anteriores, la *Asset Library* en la que los usuarios suben de manera gratuita diferentes recursos que pueden ser muy útiles para el desarrollo, como modelos 3D, colecciones de *sprites* o *scripts* que aportan varias funcionalidades tanto al motor como a lo que estés creando con él.

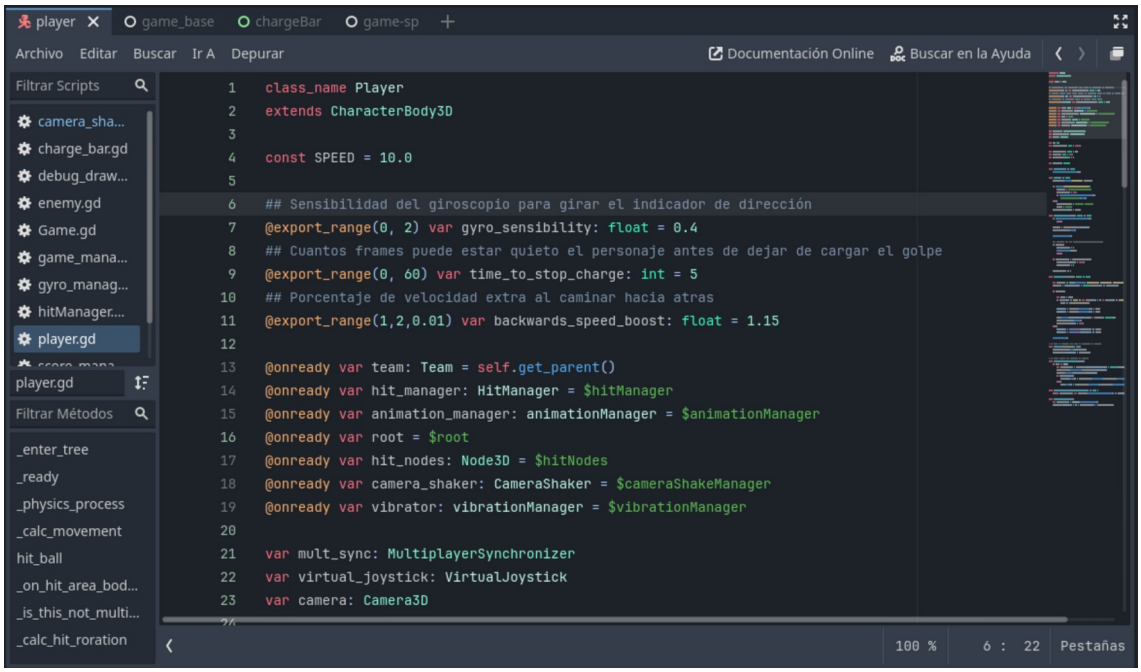

Figura 18: Editor de código del editor de *Godot*

Por último, en la parte inferior de la pantalla tenemos un apartado con cinco pestañas. La primera de ellas, salida, nos muestra toda la información que tanto el motor como tu propio proyecto quiere mostrar; cada vez que se hace utiliza la función *print* el resultado se muestra aquí. La siguiente pestaña es el depurador, una herramienta tan compleja que tiene sus propias nueve pestañas, y que sirve para utilizar todas las herramientas de depuración del motor, como ser capaz de parar la ejecución en un punto concreto y examinar el contenido de todas las variables, ver los errores que han saltado durante la ejecución, ver un desglose del uso de memoria y de memoria gráfica, monitorizar ciertos parámetros de la ejecución como la cantidad de fotogramas por segundo o la cantidad de llamadas de dibujo totales o examinar todos los paquetes de red enviados y recibidos. Es una herramienta esencial para las diferentes partes del desarrollo.

Figura 19: Pestaña de perfilador del depurador del editor de *Godot*

La siguiente pestaña es la de audio, donde se pueden añadir y gestionar diferentes buses de audio, editando el volumen de las diferentes mezclas y añadiendo efectos de postprocesado. Seguimos con la pestaña de animación, en la que tenemos una línea de tiempo para crear y editar animaciones, siendo capaz de añadir diferentes propiedades de los nodos a la línea para editar sus valores a través del tiempo e interpolarlas. Por último, el editor de *Shader*[8] te permite crear y modificar todos los tipos de sombreadores con los que se puede trabajar en *Godot.* Para los sombreadores en formato *gdshader* ,el formato por defecto que utiliza este programa, el editor se comporta igual que el editor de código, pero para los sombreadores visuales el editor entra en modo gráfico, en el cual se utiliza un sistema visual para la creación de estos. Este modo está algo más limitado que el editor basado en código, pero facilita bastante la creación de sombreadores.

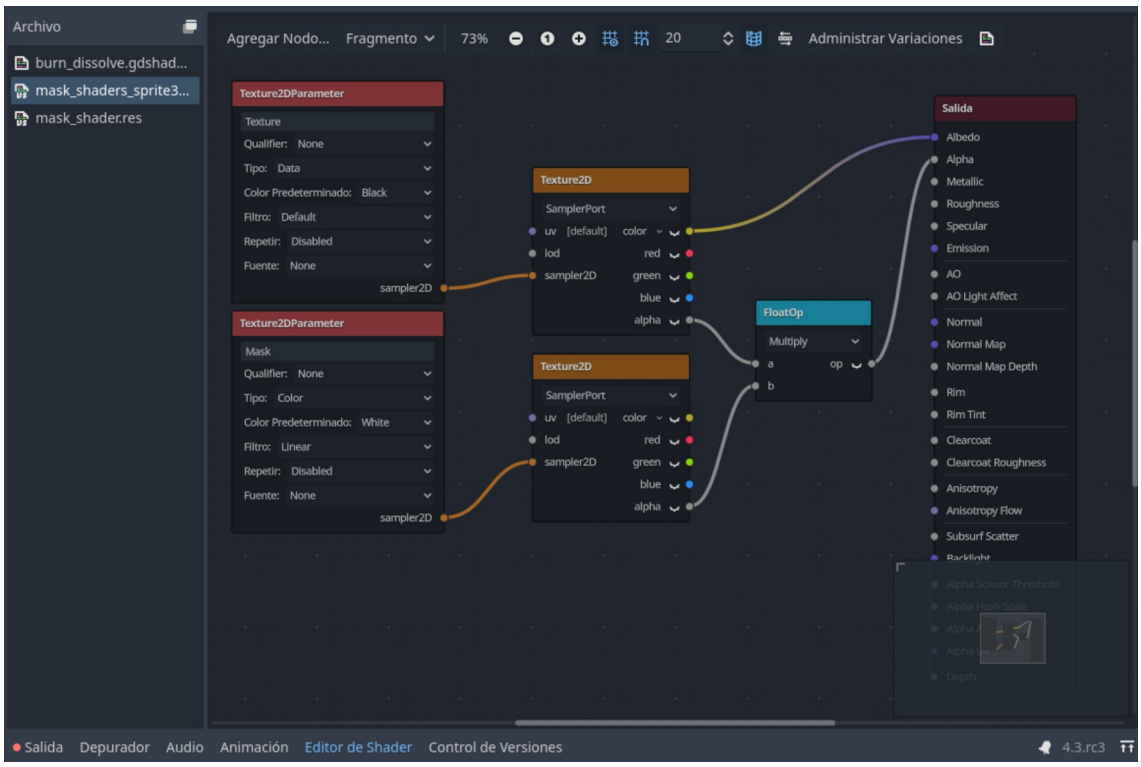

Figura 20: Editor de *Shaders* editando un sombreador visual.

2.2.2 Escenas

Tal y como se ha dicho antes, en *Godot* prácticamente todo es una escena, y estas se podrían describir como un conjunto de nodos. Para poder empezar a trabajar en el editor tiene que haber al menos una escena abierta y cuando creas una escena es simplemente un nodo del

8 El sombreador, o *shader*, es un *software* que se ejecuta en una tarjeta gráfica para determinar cómo debe dibujarse un objeto. [18] .

tipo que hayas elegido, que actúa como raíz de la escena. Si guardas un nodo o un conjunto de nodos como un archivo, estos se convierten en una escena, y las escenas pueden estar compuestas de más escenas. Este funcionamiento es muy útil para poder reutilizar un conjunto de nodos varias veces en el mismo proyecto y poder hacerle cambios en cualquier momento sabiendo que se actualizara en todas las instancias de esa escena que haya en el proyecto. Además, una escena puede heredar de otra, obteniendo todos los nodos de esta, pero pudiendo añadir más nodos. Es un sistema muy versátil y potente, aunque en un inicio puede ser algo confuso.

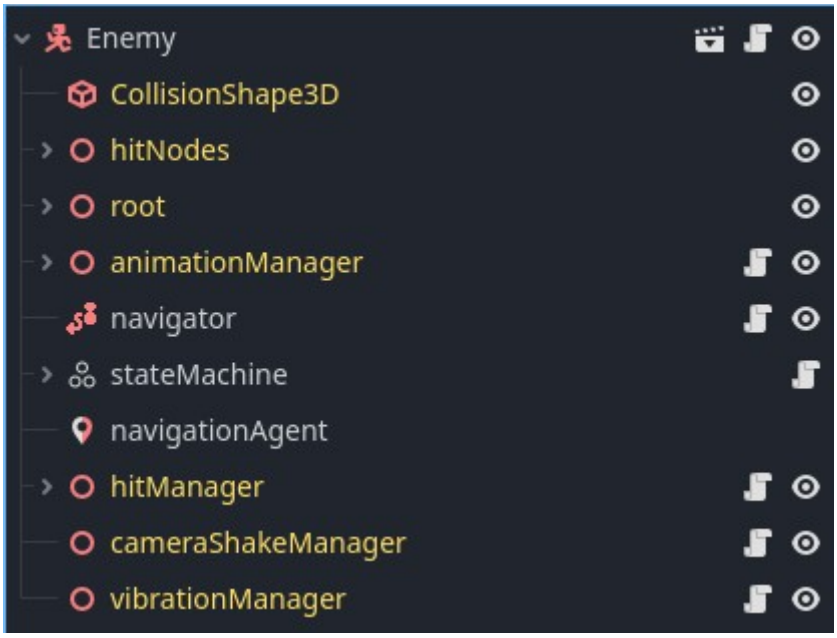

Figura 21: En una escena heredada, los nodos amarillos son los heredados mientras que los blancos son los añadidos de esta escena

2.2.3 Scripts

Los scripts son archivos con código que se añaden a los nodos para conferirles nuevas funcionalidades y son representados en la jerarquía con el icono de un pergamino. Estos scripts pueden programarse en varios lenguajes, pero por defecto vienen preparados para utilizar *GDScript* un lenguaje orientado a objetos, con una sintaxis similar a *Python* en el que el código se organiza utilizando indentación y que utiliza tipado gradual, lo que significa que está a medio camino entre el tipado estático y dinámico. Lo más básico sobre este lenguaje sería lo siguiente:

Al inicio de un *script* puedes, de manera opcional, definir una clase para el script que estás creando, haciendo así que constituya una nueva clase y pueda ser creado desde el editor y referenciado desde otros scripts. Después tienes que indicar desde que clase extiende el *script* que estás creando, lo que indicara que variables y funciones heredara; las clases y los tipos de nodos son lo mismo, así que sí añades un *script* a, por ejemplo, un nodo *navigationAgent*, automáticamente el *script* extenderá de la clase *navigationAgent*.

Lo siguiente que se puede definir en el *script* son las variables, que pueden ser variables normales, constantes o enumeraciones. Cuando defines una variable puedes asignarles de manera opcional una clase y un valor inicial, y en caso de no asignarlos el compilador los

asignara en tiempo de ejecución dependiendo del contexto. Se puede programar perfectamente sin asignar tipos a las variables, pero es una buena práctica hacerlo, ya que permite al editor ofrecer mejores recomendaciones de autocompletado y reconocimiento de errores, es más claro para la persona que está leyendo el código y ayuda a ahorrar recursos a la hora de compilar el código en tiempo de ejecución. Hay dos anotaciones que se pueden añadir al inicio de la definición:

- @export te permite exponer la variable al editor para poder editarla sin tener que entrar al código. Se le pueden añadir varias opciones para limitar el tipo de datos que se pueden asignar a la variable, como solo números en cierto rango o archivos de una extensión concreta.

- @onready asignará la variable al valor que le hayas indicado en el momento de ejecución de la función *_ready*, de la que hablaremos más adelante. Es muy útil cuando quieres asignar valores que solo estarán definidos una vez cargada la escena, como por ejemplo referencias a otros nodos de la escena.

Algunos ejemplos de definición de variables podrían ser los siguientes:

```
1 @export var goal_to_score: Node3D
2 @export_range(1,2,0.01) var backwards_speed_boost: float = 1.15
3 @onready var team: Team = self.get_parent()
4 var number = 5
5 var another_number: int
```

Después de definir las variables del *script* podemos crear métodos, que funcionan como en cualquier otro lenguaje. El tipo de las variables que le pasamos a las funciones se puede indicar de manera opcional, y lo mismo con el tipo que se devuelve. Hay algunos métodos virtuales que están reservados para ejecutarse en momentos concretos de la ejecución y son imprescindibles a la hora de programar funcionalidades básicas. Los más importantes son:

- _init(): Es el constructor de la clase y se ejecutará cada vez que se instancia la clase.

- _ready(): Se ejecutará en cuanto el nodo que la ejecuta y todos sus hijos se hayan cargado en la escena

- _process(delta): Se ejecuta en cada fotograma en el que el nodo está activo, el parámetro delta indica el tiempo que ha pasado desde que se procesó el último fotograma. Esta función es esencial para cualquier proceso que se tenga que hacer de forma continuada en el tiempo.

- _physics_process(delta): Igual que el anterior, pero en vez de procesarse cada fotograma se procesa cada cierto tiempo. Se utiliza para programar procesos sin que estos dependan de la cantidad de fotogramas por segundo a la que se está ejecutando el programa.

Todos los operadores y funciones como «*if*» y «*for*» funcionan igual que en la mayoría de lenguajes de programación, con algunas diferencias menores en los nombres y la adición de operadores en inglés como «*not*» o «*or*», que son equivalentes a «!» y «||» respectivamente.

Godot te permite cargar un *script* automáticamente cuando se inicie el juego y hacer que este se mantenga siempre activo de fondo. Este sistema se llama a*utoload*, e implementa el patrón *singleton,* en el que existe una única instancia de una clase y siempre que se tenga que acceder a la clase se hace desde esa instancia. Con *autoload* cualquier nodo desde cualquier escena y lugar de la jerarquía puede acceder a las clases que haya cargado.

2.2.4 Señales

Las señales son un sistema propio de *Godot* para enviar información entre nodos en diferentes puntos de la jerarquía del árbol, muy útiles para, por ejemplo, hacer que un hijo mande ejecutar una función a un nodo padre sin tener que obtener una referencia a este primero. Además de poder crear muestras propias señales e indicar cuando se emiten, todos los nodos incluyen una serie de señales que se emiten en momentos específicos y son la manera principal de comunicación entre nodos. Por ejemplo, cuando un *Area3D* detecta que un cuerpo ha entrado en su área, emite la señal *body_entered*(body), que si ha sido previamente conectado a algún script, nos permitirá ejecutar código cuando se detecte la colisión y nos permitirá acceder al cuerpo con el que se ha colisionado. Existen una enorme variedad de señales en cada nodo para una gran cantidad de situaciones, y en caso de que no exista una para un caso de uso concreto se puede crear.

2.2.5 Ejecución

Al contrario que en otros motores, en *Godot* la ejecución del juego no se hace dentro del propio editor, sino que se crea un nuevo proceso en el que se ejecuta el juego. Esto tiene la desventaja de que hace algo más complicado interactuar con el juego para editar valores de nodos mientras se está ejecutando, pero a cambio la ejecución es más similar a la de un programa ya exportado, lo que hace sea mucho más difícil encontrar errores que solo suceden en la versión exportada. Esto también facilita enormemente la depuración remota, que es increíblemente útil en este proyecto, ya que todas las pruebas se hacen en un móvil *Android*: en *Godot* con darle aún botón el juego se exporta y se instala directamente en el teléfono en cuestión de segundos, con las mismas opciones de depuración que si estuviera ejecutándose en el ordenador. De no ser por esto el desarrollo habría resultado mucho más complicado.

2.2.6 Conexiones de red

Godot ofrece diferentes funciones y nodos para implementar comunicación de red en tus videojuegos. Tiene nodos que son abstracciones de alto nivel de las funciones de red para facilitar la conexión directa entre diferentes dispositivos que funcionan haciendo uso de la *MultiplayerAPI,* la principal abstracción para conexiones de red de *Godot* que incluye señales que se lanzan cuando se abren y se cierran nuevas conexiones, maneja el envío de los paquetes que generan el resto de nodos y se encarga de mantener abiertas las conexiones con los pares a los que este conectado.

Utilizando esta *API* se puede hacer uso de las llamadas a procedimientos remotos, o RPC, que son funciones que pueden ser llamadas de manera remota, es decir, por otro par de la conexión. Estas funciones pueden crearse en cualquier momento utilizando la anotación «@rpc» y se pueden configurar diferentes parámetros, como quien puede llamarla, en que clientes se debe ejecutar o el tipo de paquetes que se utilizaran para la conexión. La API

también se utiliza en el nodo *MultiplayerSynchronizer*, que sirve para mantener cualquier variable sincronizada entre todos los pares a los que la API esté conectada.

Además, *Godot* también te permite trabajar a bajo nivel, facilitándote funciones para enviar paquetes de red tanto en TCP como en UDP para crear funciones de red que no existan en *Godot* o para tener un control más preciso y granular de la información que se envía y como se hace.

Aunque no está pensado principalmente para videojuegos, *Godot* incorpora nodos y funciones para realizar conexiones HTTP e incluso HTTPS, incluyendo soporte para certificados SSL, *Web Socket* y *WebRTC.*

2.3 Multijugador

2.3.1 Introducción

Cuando hablamos de multijugador en videojuegos nos referimos al hecho de jugar varias personas juntas al mismo juego compartiendo el mismo mundo virtual. Hay dos maneras bien diferenciadas en las que se implementa el multijugador en los videojuegos.

La primera sería lo que comúnmente se conoce como pantalla partida, en la que varias personas pueden jugar en el mismo dispositivo utilizando cada uno un mando. En este modo lo más normal es que se reserve un trozo de la pantalla para cada jugador, de ahí el nombre, aunque dependiendo del tipo de juego esto no es necesario. No hace falta ningún tipo de conexión a internet para este tipo de multijugador y el juego se encarga de separar la entrada de cada mando en diferentes jugadores.

La segunda manera se conoce como multijugador en línea, en el que cada dispositivo representa a un jugador y estos se conectan vía internet para jugar juntos. Es necesaria una conexión de red para poder llevarlo a cabo, aunque no tiene por qué estar conectada a internet si todos los jugadores se encuentran dentro de la misma red, ya sean conectados por cable o vía wifi: esto es lo que se conoce como conexión LAN o de área local.

En este proyecto nos centraremos en el multijugador en línea, ya que el formato móvil dificulta mucho la creación de multijugador a pantalla partida por la falta de espacio en la pantalla tanto a nivel de visibilidad como de controles.

2.3.2 Implementaciones

Al nivel más esencial el multijugador en línea se basa en intercambiar paquetes con información sobre lo que está pasando en el juego entre los diferentes clientes. Dependiendo del tipo de juego con el que se esté trabajando el contenido o la manera en la que se envían estos paquetes puede cambiar enormemente. En juegos por turnos, por ejemplo, el tiempo entre que se envía un paquete y este llega a su destino no es tan importante como en un juego de acción basado en movimientos rápidos. En cambio, quizás en este último que se pierda algún paquete por el camino no resulta tan dramático como podría resultar en un juego de estrategia por turnos en el que se tiene que transmitir mucha información entre los jugadores. Por esto mismo, no existe una única implementación del multijugador, ya que cada juego tiene sus necesidades y sus prioridades.

Las diferentes implementaciones del multijugador se podrían separar en dos grandes grupos, las basadas en conexiones *peer to peer* y las basadas en conexiones de cliente a servidor.

2.3.2.1 Peer to peer

Una conexión *peer to peer* (P2P), conocida en español como conexión de pares se caracteriza porque todos los involucrados se conectan directamente entre sí sin la necesidad de un servidor intermedio, y los dispositivos actúan a la vez de cliente y de servidor. Este modelo era el estándar en los inicios de los juegos multijugador y se sigue utilizando hoy en día aunque en menor medida.

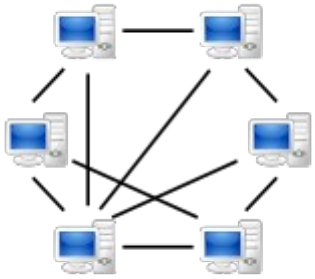

Figura 22: Ejemplo de una red basada en P2P. Fuente: https://en.wikipedia.org/wiki/Peer-to-peer

Este modelo funciona especialmente bien en redes de conexión local, ya que evitas los posibles cortafuegos y sistemas de seguridad que dan problemas a la hora de conectar dos dispositivos directamente y elimina en gran parte la latencia[9] entre clientes, que puede llegar a ser muy problemática en este tipo de redes. Existen varias maneras de implementar este sistema, ya que se puede hacer que un dispositivo tenga autoridad sobre el resto y actué como un servidor o se puede hacer que todos compartan la responsabilidad a partes iguales. Para que funcione correctamente, todos los dispositivos tienen que estar compartiendo constantemente entre sí la información necesaria para el correcto funcionamiento de la partida.

Las principales ventajas que tiene este tipo de multijugador es que, en comparación con sistemas que utilizan un servidor central, tienden a tener una menor latencia de red, ya que los paquetes viajan de un cliente a otro sin tener que pasar primero por ningún servidor intermedio. Además, la falta de servidores centrales hace que no haga falta tener que mantener una infraestructura de red para que el multijugador funcione, lo que libera a los desarrolladores de una enorme carga de tiempo y dinero.

Los principales problemas son que al tener que estar varios dispositivos compartiendo información todo el tiempo, si alguno de los jugadores tiene una mayor latencia que el resto, ya que tienen que esperar a recibir los paquetes de todos los jugadores, se ralentizara el juego para todos los jugadores. Además, se ha de tener mucho cuidado de mantener la

9 La latencia de red es el tiempo que se tarda un paquete desde que se envía hasta que llega a su destino. Normalmente, se mide en milisegundos y una mayor latencia significa una conexión más lenta.

sincronización entre los clientes en todo momento, ya que incluso pequeñas diferencias entre los valores pueden acabar haciendo que los cálculos entre los clientes varíen enormemente y que cada jugador acabe viendo una cosa diferente en su pantalla, rompiendo totalmente la experiencia de juego. Al tener los jugadores autoridad sobre los datos se vuelve imposible impedir que estos puedan hacer trampas modificando estos datos durante la partida utilizando herramientas externas. Por último, este sistema es muy difícil, si no imposible, de escalar para partidas con muchos jugadores, ya que la complejidad de los mecanismos para sincronizar los datos entre los jugadores aumenta mientras más gente haya que sincronizar .

2.3.2.2 Cliente y servidor

En un sistema de cliente y servidor, todos los datos están centralizados en uno o varios servidores, que reciben los datos de los usuarios, los procesan y les devuelven los datos necesarios a cada uno de ellos. Es el sistema más utilizado a día de hoy cuando se diseñan videojuegos en línea.

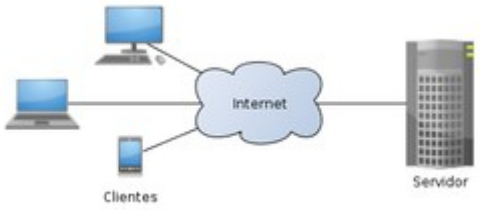

Figura 23: Un diagrama cliente-servidor vía Internet. Fuente: https://es.wikipedia.org/wiki/Cliente-servidor

Con este sistema, los clientes no ejecutan el código que procesa los datos en sus dispositivos, sino que envían esos datos, como por ejemplo podrían ser las teclas del mando que han pulsado, y el servidor le devuelve al juego esos datos procesados, en este caso la posición a la que se moverá el personaje: esto hace que los clientes no tengan ningún tipo de autoridad sobre la partida. Normalmente, los servidores procesan los datos recibidos y actualizan el estado de cada jugador cada cierta cantidad de tiempo, como por ejemplo treinta veces por segundo, por lo que los clientes solo tienen que interpolar estas actualizaciones para dar la sensación de continuidad.

Esto soluciona muchos de los problemas que tiene el acercamiento anterior porque dificulta mucho las trampas por partes de los jugadores, ya que estos no tienen acceso a como se procesan los datos. Además, como hay un servidor central que se encarga de enviar la información a todos los clientes, no habrá de-sincronización, ya que todos recibirán la misma información. Por último, como el que marca el ritmo de actualizaciones es el servidor, que un jugador tarde más en enviar y recibir datos no perjudicara al resto de clientes.

Por otro lado, este acercamiento viene con sus propios problemas, aunque todos son derivan de lo mismo: la latencia. Para poder intercambiar paquetes con un servidor, estos tienen que salir de tu dispositivo, pasar por tu ISP, llegar al servidor, ser procesados y repetir el mismo camino a la inversa. Aunque este proceso se lleve a cabo en milisegundos, hay muchos juegos en los que se puede llegar a notar enormemente. El principal problema es que, al procesar el servidor los movimientos del usuario, este puede notar muy fácilmente si el resultado de esos movimientos no corresponde con las teclas que acaba de pulsar, o en juegos donde se interactúe entre jugadores las diferencias de latencia entre ellos puede dar una enorme

ventaja a los jugadores con menor latencia, lo que rompe completamente la experiencia de juego.

Dependiendo del tipo de juego la cantidad de latencia aceptable puede variar enormemente. En juegos de disparos que dependen enteramente de los reflejos de los jugadores cualquier cantidad por encima de los 100 ms se suele considerar injugable. Una latencia alta puede ser culpa de la calidad de la conexión del jugador, de la distancia física entre jugador y servidor, o de problemas en el propio servidor, no de conexión, sino del tiempo que tarda en procesar los datos, ya sea porque no es lo suficientemente rápido o porque está sobrecargado con demasiadas peticiones [26] .

2.3.3 Mitigación de latencia

Independientemente de la implementación del multijugador, el principal problema que se suele tener es la latencia alta, por lo que los desarrolladores suelen utilizar diferentes técnicas para mitigarla.

Por ejemplo, el videojuego «*Counter Strike 2*» implementa lo que han denominado sistema de sub-ticks, en el que en vez de confiar en el orden en el que los paquetes llegan al servidor, estos paquetes contienen información sobre el momento exacto en el que han sido enviados, para que así el servidor pueda ordenarlos de manera correcta a la hora de procesarlos y pueda notar las pequeñas diferencias de milisegundos entre los diferentes paquetes sin tener que aumentar demasiado la velocidad de procesado del servidor y mitigando las diferencias de latencia entre diferentes jugadores.

Otro sistema que se ha implementado en muchos juegos durante los últimos años es el código de red retrospectivo, normalmente conocido como «*rollback netcode».* Es especialmente popular en juegos de lucha una contra uno que suelen tener sistemas de multijugador P2P. Este sistema se basa en que el juego intenta predecir los siguientes movimientos de los jugadores con un pequeño margen de tiempo de algunos milisegundos. En caso de que la predicción se cumpla, la partida siga con normalidad, y en caso de que no, la partida retrocede a un estado anterior y se continúa con los valores correctos. Estos retrocesos son de una cantidad de tiempo tan pequeña que apenas se notan durante una partida, y si el sistema está bien programado no deberían producirse demasiado [27] . Si funciona correctamente, elimina por completo la sensación que produce la alta latencia a los jugadores, aunque puede llegar a ser muy complejo de programar.

2.3.4 Selección

Para este proyecto se diseñará un sistema de multijugador basado en conexiones peer to peer, ya que cumple con las necesidades del proyecto: será un juego de un jugador contra otro, así que solo habrá dos dispositivos entre los que haya que sincronizar los datos y eliminará todo tipo de mantenimiento de servidores, lo que podría haber sido bastante problemático. Por la naturaleza de este proyecto, la protección contra tramposos no es una prioridad como si lo sería en un producto comercial, y el multijugador está centrado principalmente en jugar en red local, lo cual es idóneo para este sistema.

Se tendrá que trabajar en maneras de evitar los problemas de sincronización y en como habilitar el juego por internet.

2.3.5 TCP y UDP

Cuando se envían paquetes de red hay dos protocolos que se pueden utilizar para el transporte, TCP y UDP.

TCP asegura que los paquetes siempre llegarán de manera confiable y en orden, pero la latencia generalmente es más alta debido a la corrección de errores. También es un protocolo bastante complejo porque entiende lo que es una "conexión" y se optimiza para objetivos que a menudo no se adaptan a aplicaciones como los juegos multijugador. Los paquetes se almacenan en el búfer para enviarlos en lotes más grandes, intercambiando menos sobrecarga por paquete para una mayor latencia. Esto puede ser útil para cosas como HTTP, pero generalmente no para juegos.

UDP es un protocolo más simple que solo envía paquetes (y no tiene ningún concepto de "conexión"). No poseer corrección de errores lo hace bastante rápido (baja latencia), pero los paquetes pueden perderse en el camino o ser recibidos en el orden incorrecto. Además de eso, el MTU (tamaño máximo de paquete) para UDP es generalmente bajo (solo unos pocos cientos de bytes), por lo que la transmisión de paquetes más grandes significa dividirlos, reorganizarlos y volver a intentar si una pieza falla [28] .

Cuando se desarrolla una aplicación de red se tiene que tener muy en cuenta que protocolo se utilizara y para que, ya que pueden utilizarse los dos en conjunto para enviar diferentes paquetes según las necesidades de los datos. Por ejemplo, en un videojuego podría utilizarse UDP para enviar las acciones de los jugadores y TCP para los mensajes del chat.

3 Desarrollo

3.1 Preproducción

3.1.1.1 Conceptualización

Antes de poder empezar siquiera con el proceso de preproducción hay que tener una serie de ideas iniciales sobre como será el juego y hacia donde se quiere llevar. Hasta ahora, las ideas planteadas se pueden resumir en: un juego de móvil ligero y de partidas rápidas con mecánicas sencillas. Todas las posibles ideas para el desarrollo tienen que partir de esta base.

Desarrollar un juego para móviles trae ciertas limitaciones, principalmente en el control del juego. Los móviles no tienen botones, así que todo el *input* del jugador este relegado a la pantalla táctil, por lo que muchos juegos optan por crear botones virtuales y colocarlos en distintos puntos de la pantalla para simular el esquema de control de un mando de consola tradicional. Ese enfoque no me convence especialmente, porque estos botones suelen ser bastante incómodos y poco precisos, especialmente si es un juego rápido o que dependa de acciones muy precisas. El objetivo es que el juego sea sencillo de controlar y que esté bien adaptado a los controles táctiles de un móvil, en vez de intentar imitar un esquema de control más tradicional, por lo que impuse la limitación de que el juego se pudiera controlar con una sola mano. Esto limita enormemente el tipo de juego que se puede hacer, pero también te obliga a ser más creativo con las diferentes soluciones para los controles.

Figura 24: Interfaz de «*PUBG Mobile*» con todos los botones activados al mismo tiempo. Fuente: https://www.androidcentral.com/gaming/android-games

Para asegurar que las ideas planteadas no fueran demasiado amplias para el alcance de este proyecto, la búsqueda de ideas se planteó del siguiente modo: al estar aprendiendo como utilizar *Godot* y estar buscando información sobre desarrollo de videojuegos, cuando encontraba una tecnología o una mecánica interesante que veía capaz de implementar, empezaba a construir la idea basándonos en ella. La decisión final quedó entre las dos siguientes ideas:

La primera idea vino de estudiar los algoritmos de colapso de función de onda. Estos funcionan tanto en dos como en tres dimensiones y son algoritmos de generación a los que se les puede dar una serie de normas que seguirán para generar un conjunto de píxeles o de polígonos. Se suele usar en videojuegos para generar escenarios de manera pseudoaleatoria siguiendo algún tipo de lógica [19] . Con esto en mente, la idea se basaba en un juego del género *idle* o *clicker*, que se caracterizan por qué el jugador tiene que hacer acciones muy simples como tocar la pantalla repetidamente para obtener un recurso que puede invertir en métodos para obtener este mismo recurso más rápido y de forma más eficiente, llegando a un punto en el que la interacción del jugador con el juego es mínima [20] . En este caso, el recurso serían piezas de una construcción, como por ejemplo un pueblo o un castillo, que se construiría automáticamente utilizando uno de los algoritmos anteriormente mencionados. Normalmente en este tipo de juegos el principal atractivo para avanzar en él es ver los propios números crecer hasta que llegan a cifras absurdas; en este caso, la construcción sería la que crecería de esta manera.

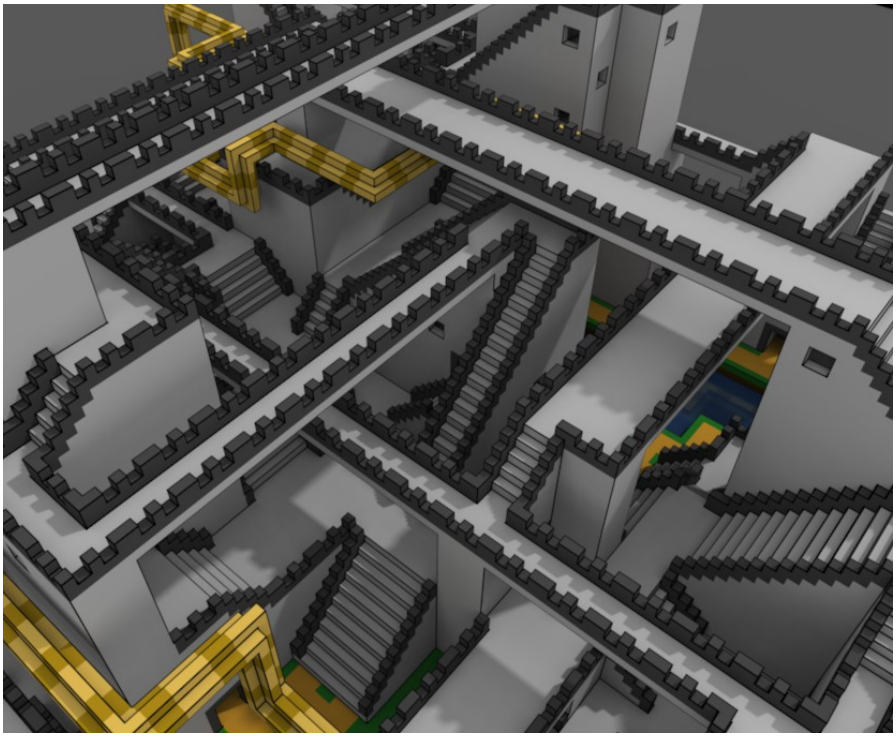

Figura 25: Conjunto de pasillos y escaleras generados con una algoritmo de colapso de función de onda. Fuente: https://github.com/mxgmn/WaveFunctionCollapse

Por un lado, es muy sencillo a nivel mecánico, pero a nivel de diseño sí que presenta algunas dificultades: hay que diseñar muy bien el ratio al que crece la construcción y cuanto puede alterar el jugador ese crecimiento para no aburrir al jugador con un crecimiento demasiado lento, pero tampoco abrumarlo con uno demasiado rápido. Además, a nivel visual, habría que trabajar mucho las normas del algoritmo para que produjera diseños interesantes de ver y que no se volvieran repetitivos después de unas pocas horas de juego. Por último, las piezas de construcción tendrían que ser modelos en 3D visualmente interesantes y que encajaran lo suficientemente bien entre ellos como para que no se notaran las costuras.

La otra idea vino de pensar en cuál sería el esquema de control que mejor aprovechase las características de un móvil. En un móvil tienes una pantalla por la que arrastras uno o varios dedos, así que se podría hacer un juego en el que el esquema de control estuviera basado en dibujar líneas arrastrando los dedos. Hay varios juegos con esta premisa, como «*Kirby: Canvas Curse*» (2005) de Nintendo DS, en el que el protagonista es una bola y tienes que trazar los caminos por los que rodara, o la saga «I*nazuma Eleven*», en el que juegas partidos de futbol dibujando el camino que tomaran los jugadores por el campo.

Figura 26: «Kirby Canvas Curse», 2005, Nintendo DS. Fuente: https://www.ign.com/games/kirby-canvas-curse

Figura 27: «Inazuma Eleven», 2011, Nintendo DS. Fuente: https://www.ign.com/games/inazuma-eleven

Fue este último la inspiración para esta idea en la que llevarías un equipo de tres jugadores a los que controlarías dibujando los caminos que han de seguir. Mientras tú no controlas a un jugador, este tendría su propia inteligencia artificial encargada de colocarlo en puntos estratégicos para ayudarte con las jugadas o mandarlos a defender cuando fuera necesario. Mecánicamente, esta idea es más interesante que la anterior, pero también más difícil de llevar a cabo, ya que hay combina un método de control que puede ser difícil de implementar con una inteligencia artificial capaz de coordinar a los dos jugadores de tu equipo, y a los tres del equipo rival. El juego podría hacerse tanto en dos como en tres dimensiones, y el apartado artístico no tendría por qué ser especialmente complejo.

Al final, la idea ganadora fue la segunda, porque parecía más interesante tanto a nivel de desarrollo como para el juego final, y aunque tiene puntos que pueden resultar problemáticos, parece posible llevarla a cabo.

3.1.2 Prototipado

Con la idea ya clara, el siguiente paso es hacer un prototipo en el que implementar las mecánicas básicas para poder comprobar que tal funcionan en un entorno real y verificar la viabilidad del proyecto. Para este prototipo, el objetivo es que se pueda jugar una partida de principio a fin. Si todo sale bien, se continuará con el prototipo (muchas veces se vuelve a empezar de cero el juego, volviendo a implementar las mismas mecánicas, pero aprovechando el conocimiento para evitar errores que podrían dar problemas al largo plazo) para construir el juego final.

El prototipo comenzó siendo en dos dimensiones, con tres personajes en pantalla representados por el *sprite* por defecto de *Godot*, el logo del programa, y el objetivo era hacer que estos personajes se moviesen siguiendo una línea dibujada por el jugador. Por suerte, el motor incluye varias funciones para detectar toques y arrastres en una pantalla táctil, que quedan registrados como «eventos». Además, como añadimos a estos jugadores un área de colisión, son capaces de detectar cuando se pulsa con el dedo encima de ellos, así que combinamos esto con la capacidad de detectar donde y cuando el jugador está arrastrando el dedo y podemos saber si se está dibujando una línea que tiene como origen uno de los jugadores, y los puntos de la pantalla por los que está pasando el dedo.

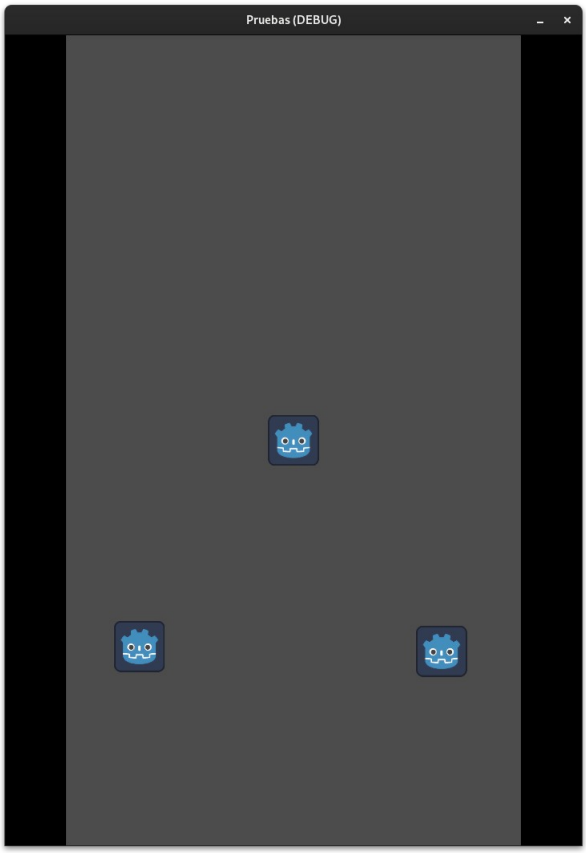

Figura 28: Prototipo con los jugadores estáticos

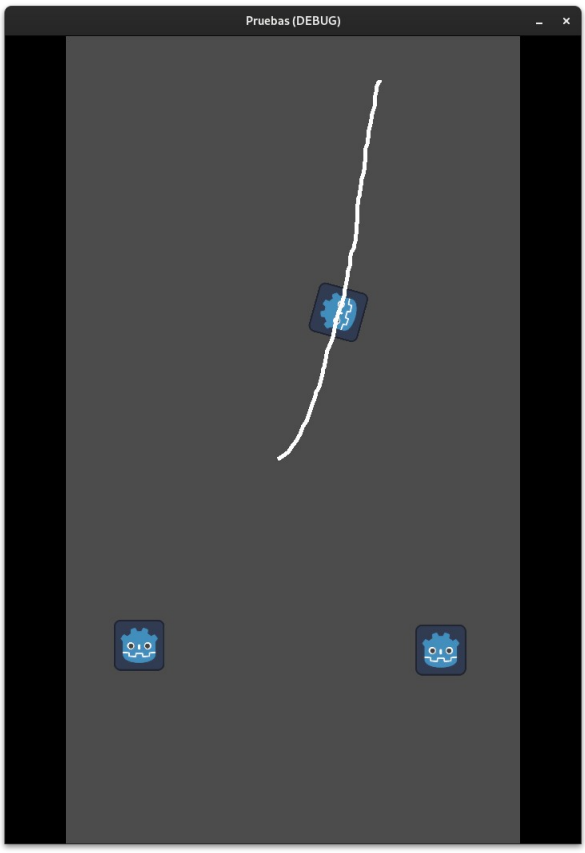

Figura 29: Prototipo con un jugador siguiendo la linea dibujada

Cada jugador se encarga de detectar los eventos de las pulsaciones, así se simplifica el manejo de varias líneas a la vez. Para dibujar la línea, se crea un objeto de tipo *Line2D, s*e le van indicando programáticamente los puntos por los que pasa el dedo y el motor se encarga de dibujar la línea. Para mover a los jugadores por esta línea, existen los nodos *Path2D* y *PathFollow2D*: el primero es simplemente un conjunto de puntos que forma un camino, así que podemos utilizar los puntos de la línea dibujada para crearlo, el segundo se dedica a seguir el camino que indica el primero, pudiendo controlar la velocidad y el avance con varios parámetros. Se han tenido que hacer algunos ajustes matemáticos para que los puntos de la pantalla correspondan con puntos del escenario en los que poder dibujar la línea.

Esta primera prueba del sistema de control parecía estar funcionando, pero empezaba a dar la sensación de que hacer el juego en dos dimensiones podría traer varias limitaciones a la hora de diseñar el resto del juego, principalmente a la hora de mover la pelota por el campo, así que de deicidio volver a programar la misma mecánica pero en 3D. Resulto ser mucho más complicado de lo esperado, por varios motivos: primero, las matemáticas para mapear un punto de la pantalla a un punto del escenario son más complicadas en un entorno en tres dimensiones y una cámara con perspectiva, y segundo, dibujar una línea por pantalla en un entorno en tres dimensiones es problemático, porque hay que elegir entre varios acercamientos diferentes. Se puede seguir dibujando una línea en dos dimensiones en la pantalla, y proyectará sobre el suelo del campo, se puede dibujar en la pantalla y mostrar por encima de los gráficos 3D, como si estuviera en otra capa, o se puede generar una línea en tres dimensiones a base de polígonos.

La última opción es la que mejor queda a nivel visual con el resto de elementos en pantalla, pero el motor no incluye ninguna manera de hacer esto. Por suerte, el usuario de *github* «*belataras»* había creado y publicado en la librería de *assets* de *Godot* un sistema para dibujar líneas en tres dimensiones. Hubo que modificar parte del código de este sistema, principalmente como se calcula el ángulo entre diferentes puntos de la recta para que funcionara correctamente para el caso de uso deseado.

Para saber en qué puntos había que dibujar la recta no se podía utilizar el mismo sistema de eventos que en la versión en dos dimensiones, ya que estos devuelven las coordenadas de la pantalla donde se ha pulsado, y el juego necesita las coordenadas del propio mundo de juego, y los diferentes intentos de hacer la conversión entre los dos con matemáticas habían fallado. La solución fue usar *raycasts*, un concepto que incluyen muchos motores gráficos, consistente en disparar un rayo desde un punto de origen a una dirección y distancia determinados para que devuelva información al golpear sobre algún objeto. En este caso, se dispara desde la cámara en dirección al punto donde se ha tocado la pantalla para que cuando el rayo golpee con el suelo del campo devuelva el punto exacto donde hay que pintar la línea. Con las coordenadas listas, se usa el mismo sistema que antes para mover al jugador.

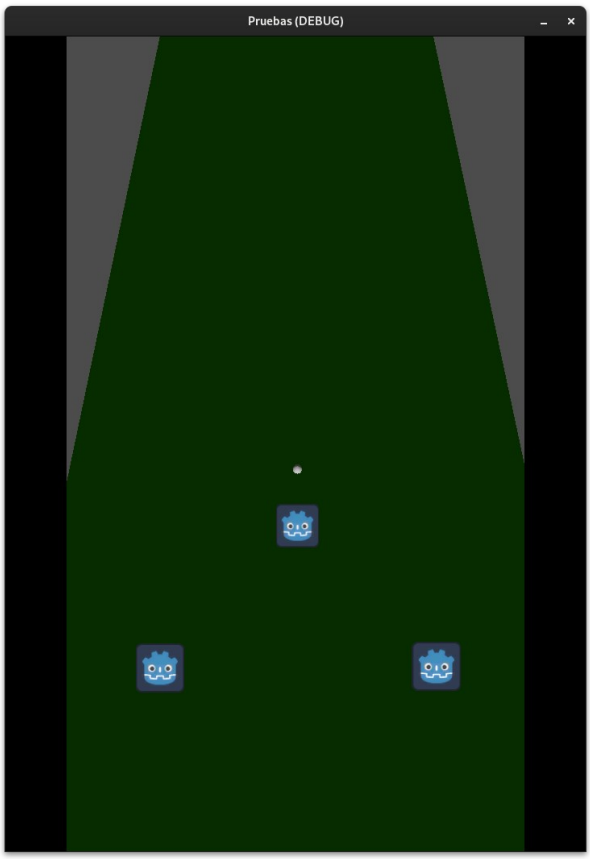

Figura 31: Segunda iteración del prototipo con los jugadores estáticos

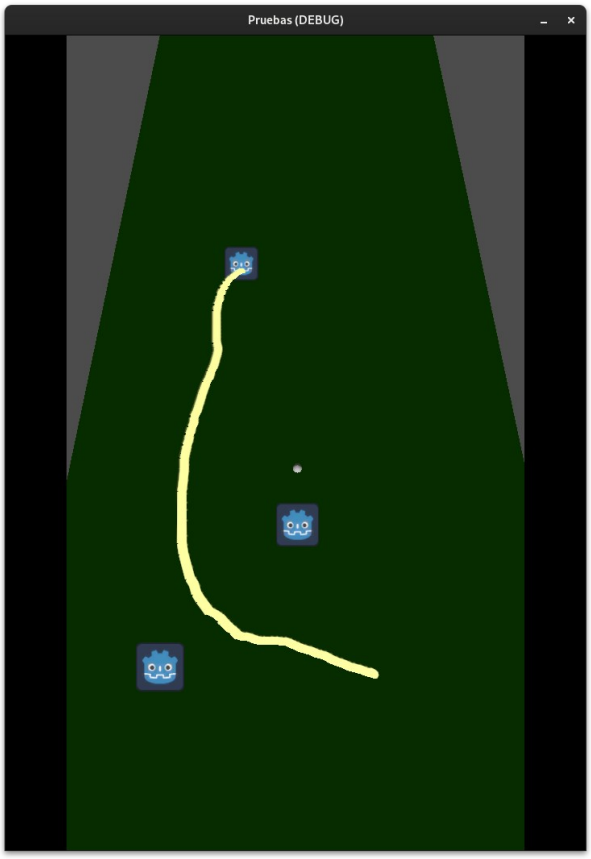

Figura 30: Segunda iteración del prototipo con un jugador siguiendo la linea

A este sistema se le tuvieron que hacer una variedad de retoques para que funcionara correctamente, como gestionar el número de puntos y la distancia entre los puntos de la línea para asegurar una velocidad de movimiento constante, varias comprobaciones constante de donde están colocados los dedos que tocan la pantalla y qué jugador se está moviendo en cada momento para evitar confundir al sistema de movimiento.

Con el paso al 3D se mantuvieron los *sprites* utilizando el nodo *Sprite3D* con la idea de hacer una mezcla entre escenario en tres dimensiones y jugadores dibujados en dos dimensiones, pero en aras de la sencillez se acabo cambiando por modelos 3D. Me descargué un modelo creado por *GDQuest*, un maniquí con una serie de animaciones predeterminadas para facilitar el proceso de prototipado.

Con el movimiento funcionando, quedaba otra pieza fundamental para poder jugar una partida, la pelota. En un principio, la pelota detectaba las colisiones con los jugadores y se adhería a ellos, pero este enfoque resulto ser bastante problemático, así que se decidió hacer al revés, que los jugadores detectaran la pelota y se la adhirieran. Cuando un jugador tiene la pelota, las colisiones de esta se desactivan para que ningún jugador pueda tocarla, y se vuelven a activar al chutar. Los jugadores tienen un temporizar que se activa al chutar la pelota y hace que no puedan volver a detectarla hasta medio segundo después de chutar, para evitar problemas con la pelota enganchándose al jugador después de chutar. Para decidir donde se lanza la pelota se da un toque en la pantalla, y se calculan las coordenadas utilizando el método del *raycast.*

Para mover la pelota del jugador al punto deseado, se utilizan *tweens*, animaciones que se pueden crear desde el código con diferentes parámetros a configurar.

Con estos dos sistemas funcionando el siguiente paso era empezar con la inteligencia artificial de los jugadores. El plan es que cuando los jugadores no estén siguiendo el camino marcado por el jugador se muevan independientemente basándose en su posicionamiento en el campo y la posición tanto de la pelota como de los enemigos. Para gestionar este comportamiento creamos una máquina de estados, con un nodo padre para gestionar el cambio entre estados, y nodos hijos para cada uno de estos. En un inicio se crean tres estados:

- *«Idle»*, en el que los jugadores están quietos esperando a pasar a otro estado.
- «Manual», cuando los jugadores están siendo controlados por el usuario.
- *«Shoot»*, para el momento en el que se chuta la pelota.

Cada estado gestiona a qué estado tiene que cambiar y cuando debe hacerlo. En este punto, el estado «Manual» es el que hace todo el trabajo para calcular el movimiento del jugador basándose en la línea dibujada. Además, cada estado tiene una animación del jugador asociada.

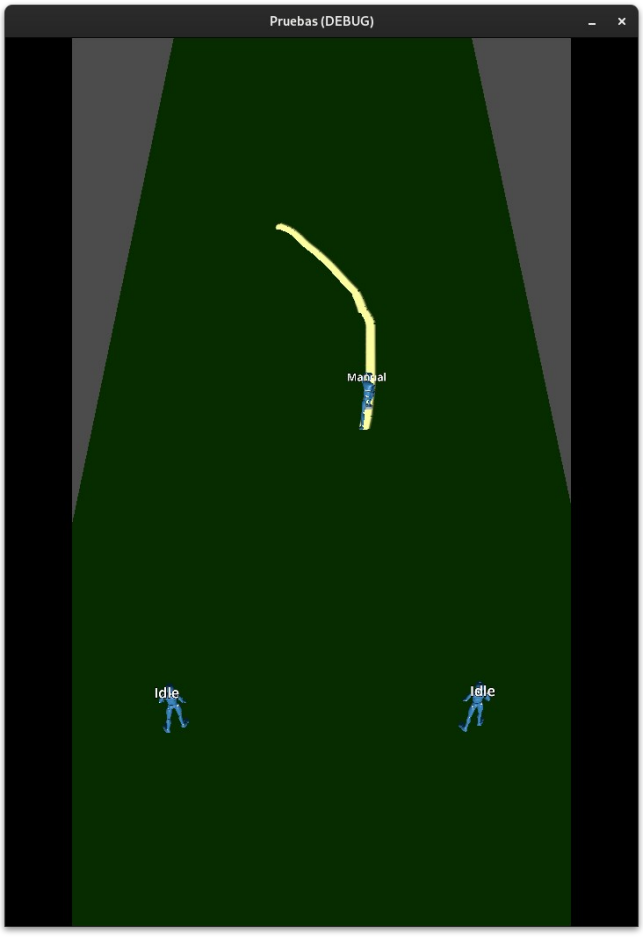

Figura 32: Una etiqueta encima de cada jugador indica su estado actual

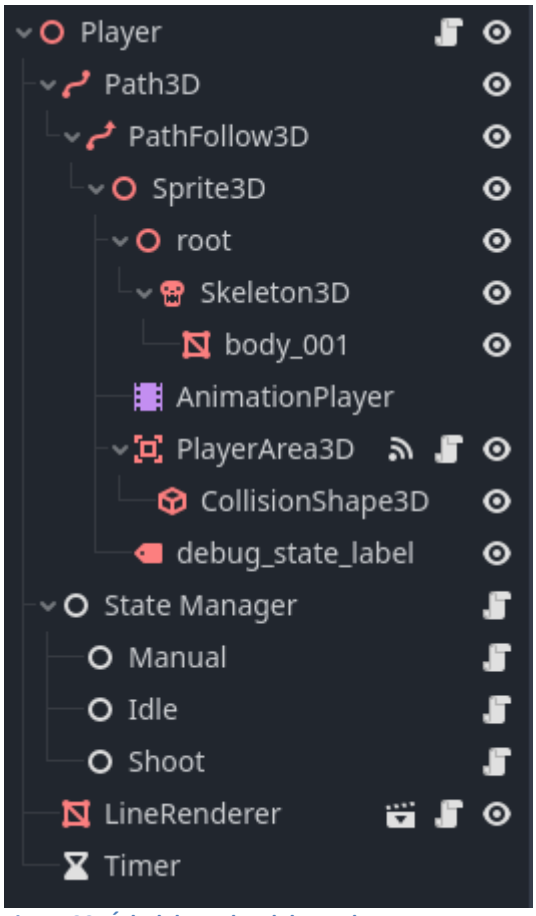

Figura 33: Árbol de nodos del Jugador

Con esta base, creamos unos jugadores rivales que compartían el mismo funcionamiento, pero sin acceso al estado «Manual», para que no pudieran ser controlados manualmente. Gracias a la herencia de los nodos de *Godot*, se pudo crear un nodo para los enemigos que heredara del nodo Jugador, y así poder reutilizar todo el trabajo hecho sin tener que repetir ni una línea de código. Después de esto, se ajustó la cámara y la perspectiva para poder enfocar a todo el campo y tener espació suficiente para maniobrar. Desde los ajustes del proyecto se cambió el ancho y largo predeterminado de la ventana, para asegurarse de que el juego se abra en la relación de aspecto correcto. Se cambió la cámara para que funcionase con una proyección ortogonal, en vez de perspectiva, además se ajustó para que independientemente de lo ancha que fuera la ventana, siempre se mantuviese igual de alta para poder adaptarse a pantallas de móviles de diferentes tamaños y se añadieron a ambos lados del campo unos cilindros para utilizarlos de portería, que detectan la colisión de la pelota. Gracias al sistema de capas y máscaras de *Godot* se pueden asignar capas de colisión a los objetos para hacer que solo ciertos objetos puedan colisionar con otros, y facilitar enormemente la detección. Por último, se añadió un nuevo estado, *«Fetch»*, que hace que el jugador vaya a por la pelota si no la tiene alguien de su equipo.

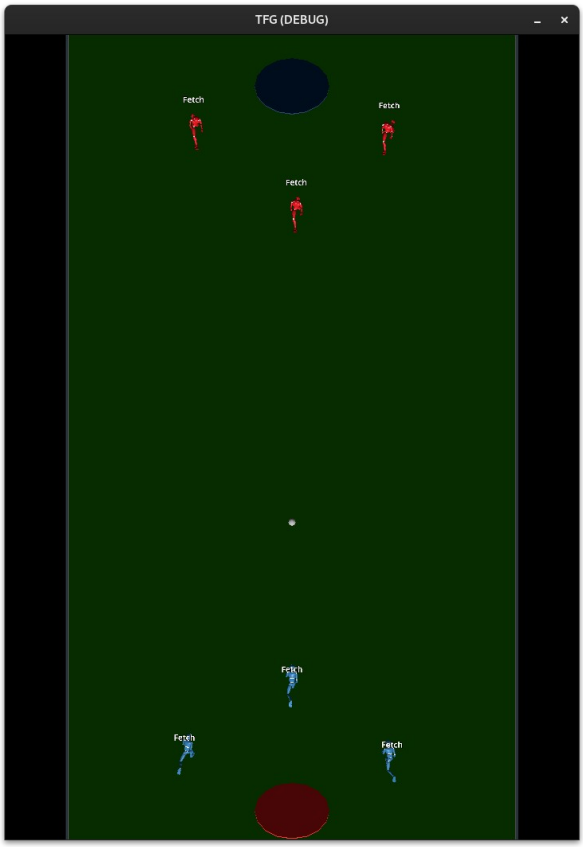

Figura 34: Todos los jugadores del partido yendo hacia la pelota

Figura 35: Jugadores del equipo rival persiguiendo al jugador con la pelota

Con estos sistemas ya implementados llegaba la parte más complicada hasta el momento, convertirlos en algo jugable. Para ser capaz de terminar una partida, hace falta que tanto los rivales como los jugadores tu equipo sepan colocarse en el campo, pasarse la pelota y avanzar para ser capaces de marcar. Además, había que implementar alguna mecánica para ser capaz de robarle la pelota a otros jugadores. Fue en este momento cuando se empezó a ver que la

complejidad de este sistema era muy alta, y durante bastante tiempo el proyecto no avanzo como debería. No solo la parte técnica se está complicando, sino que también se estaba convirtiendo en un desafío de diseño el cómo organizar a todos los jugadores a la vez de manera dinámica. Llegados a cierto punto, después de pasar algunas semanas bloqueado con como continuar expandiendo las mecánicas se decidió descartar la idea, reutilizar todo lo posible de este prototipo, y volver a empezar con una nueva idea y un nuevo prototipo.

3.1.3 Segundo prototipado

Ya que el principal problema del anterior prototipo había sido el aumento de la complejidad, tanto a nivel de planteamiento como de ejecución, esta vez el plan era algo más simple y directo, un juego de reflejos, que mantenga un jugador en cada equipo, una pelota y el mismo escenario. Pero si iba a ser un juego más rápido y directo, el sistema de movimiento a través de líneas ya no tenía lugar allí, así que se decidió descartar ese sistema que tantas horas había costado programar y en cambio el movimiento se controlaría con un joy*stick* táctil. Por suerte en la *AssetLib* encontré *Godot Virtual Joystick* por el usuario «MarcoFazioRandom», que hace exactamente lo que su nombre indica, funciona perfectamente y tiene multitud de opciones para ajustarlo a tu gusto.

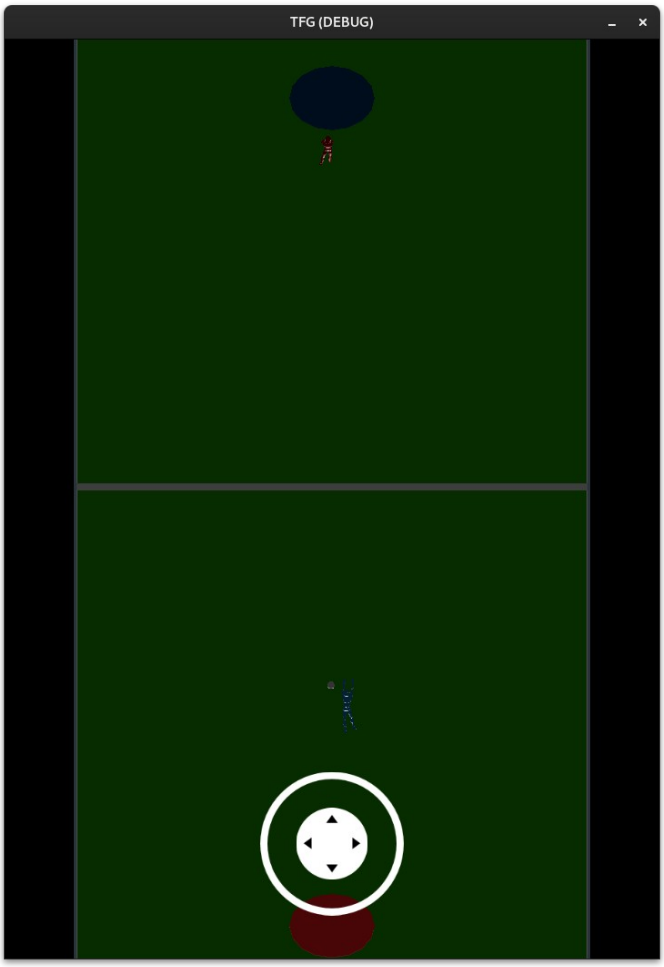

Figura 36: Primera iteración del nuevo prototipo

Este nuevo esquema de control introduce un nuevo problema, y es que al sostener el teléfono en vertical, y utilizar el pulgar para manejar el *joystick*, toda la mano queda ocupada y no hay posibilidad de añadir más botones si queremos seguir manteniendo el juego jugable con una sola mano. Al pensar en esto, se me ocurrió como podría funcionar este prototipo: al mover el personaje, este se prepara para golpear, y en el momento en el que dejas de moverte y retiras el dedo de la pantalla, este golpea. Es una idea reminiscente de los juegos de acción donde tienes que pulsar un botón en el momento justo para defenderte de un ataque, pero en este caso lo haces para golpear una pelota que viene hacia ti a gran velocidad.

Con esta idea en la cabeza, otra idea se me vino a la mente: si iba a ser un juego de reflejos y habilidad de un jugador contra otro, tendría mucho sentido añadirle un modo multijugador. Con esto decidido, elimine todo lo que no se fuera a utilizar del proyecto (todo excepto el campo de juego, el modelo del personaje y la pelota) y comencé con el segundo prototipo.

Lo primero que había que comprobar en este nuevo prototipo era si iba a ser posible integrar el multijugador. Para ello comencé con la documentación oficial sobre multijugador y los videos de «*BatteryAcidDev*», un canal de *YouTube* con una serie de videos muy informativos sobre *networking* en Godot [21] [22] .

El primer paso para un multijugador funcional es ser capaz de conectar dos clientes que estén ejecutando el juego y que sean capaz de intercambiar paquetes de red. Para ello se comenzó con la creación de un l*obby*[10]. En esta sección no se entrará en detalles sobre la implementación, ya que esto será tratado en un apartado posterior. Para poder acceder a este se creó un botón para crear una partida, y otro para unirse, junto con un campo para introducir el nombre que se mostrara en el *lobby* y otro para la dirección IP del jugador a cuyo *lobby* unirse.

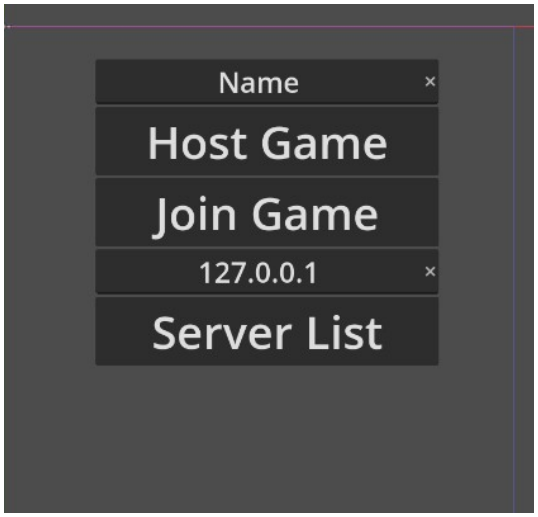

Figura 37: Menú para iniciar o unirse a una partida de multijugador

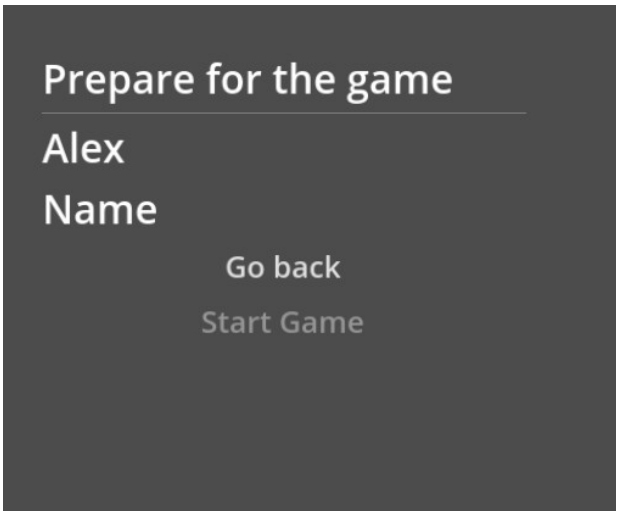

Figura 38: Vista del *Lobby* con los dos jugadores conectados

Con un sistema de conexión básico entre dos jugadores funcionando, hacía falta alguna forma de facilitar el proceso de conexión, así que se hizo una lista de servidores que escanea la red a

10 En los videojuegos multijugador un *Lobby* es una sala de espera en la que los jugadores conectados se reúnen antes de que empiece la partida. Se suele utilizar para esperar a que todos los jugadores se hayan conectado correctamente.

la que está conectado el cliente para detectar si alguien en su misma red ha iniciado una partida. Con esto funcionando correctamente, quedaba la parte más complicada del conjunto, hacer que el juego funcione en multijugador. Lo primero fue implementar dos jugadores controlables que aparecieran al iniciar la partida, separar el *input* de los dos clientes para que cada uno controle a un jugador, y sincronizar el movimiento entre ellos para que los dos jugadores estén viendo lo mismo por pantalla.

Después de esta primera implementación del multijugador parecía que todo estaba funcionando correctamente, así que avanzó a añadir las últimas mecánicas para terminar el prototipo y poder entrar en la fase de producción. Primero se añadió un indicador de carga en forma circular que se activa cuando el jugador está en movimiento, utilizando el nodo *TextureProgressBar*, que sirve para crear diferentes tipos de barras de carga en dos dimensiones. Para mostrar este indicar en un espacio en tres dimensiones primero se renderiza dentro de un *SubViewPort*, un nodo que aísla una región rectangular de una escena para ser mostrada independientemente [23] , y que puede renderizarse en una textura y mostrarse utilizando un *Sprite3D*, un nodo para mostrar texturas en dos dimensiones en un espacio en 3D.

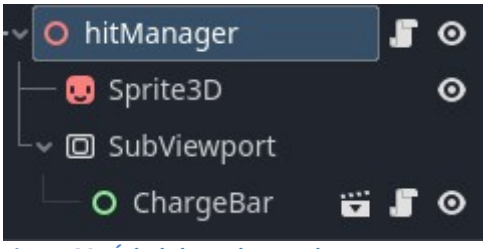

Figura 39: Árbol de nodos usados para mostrar el indicador

La barra de carga emite una señal al terminar de cargarse, que puede ser conectada a otro nodo para ejecutar una animación para golpear a la pelota.

La animación para golpear la pelota está diseñada para activar durante cierto tiempo un *Area3D* en el jugador, un nodo utilizado para detectar colisiones entre objetos. Si se detecta una colisión con la pelota, se le aplica a esta una fuerza en la dirección a la que estuviese mirando el jugador cuando la golpeo. Para indicar mejor la dirección, se añadió una línea a los pies del personaje que apunta en la dirección que mira el jugador. La pelota es un nodo del tipo *RigidBody3D*, lo que significa que está sujeto a normas físicas, a sí que al aplicarle una fuerza saldrá disparado y rebotará al golpearse contra las paredes del escenario, y si entra dentro del área de las porterías estas la detectarán y sumará un punto al jugador. Cuando se llega a tres puntos, la partida se acaba y la conexión entre los dos jugadores se cierra.

Con estas mecánicas implementadas y siendo posible jugar una partida de principio a fin, se dio por terminada la fase de prototipado y se pasó a la de producción.

3.2 Producción

En este apartado se detallará como se han llevado a cabo los diferentes pasos de la producción.

3.2.1 Diseño

Después de terminar el prototipo ya se empezaba a vislumbrar una idea clara de como quería que fuese el proyecto final.

3.2.1.1 Objetivos

Las bases sobre las que se erigiría el resto del diseño son muy claras: es un juego casual, pensado para que cualquier persona pueda entender como funciona después de jugar una partida, fácil de aprender pero difícil de dominar, tiene que poder jugarse en sesiones de juego cortas y la jugabilidad tiene que ser satisfactoria tanto jugando solo como contra otra persona. Además, tiene que poder jugarse con el móvil en vertical y utilizando solo una mano. En resumen, tiene que ser minimista. Estas bases están basadas principalmente en mis preferencias como jugador y en como yo entiendo que deberían ser los juegos para móviles de este tipo.

Con estas bases encima de la mesa, y basándome en diferentes pruebas que se han hecho a la lo largo del proceso de desarrollo, aquí están las directrices de diseño que se han seguido.

3.2.1.2 Idea general

Tanto a nivel conceptual como a nivel jugable el juego es muy simple: hay un campo simétrico verticalmente y partido en dos, con un jugador y una portería a cada lado y una pelota en el centro. Cada jugador tiene un bate con el que golpea la pelota hasta meterla dentro de la portería de su rival. Si la pelota golpea los bordes de este escenario no se saldrá sino que rebotara. Se podrá jugar con una persona jugando contra la maquina o dos jugadores enfrentándose entre ellos cada uno desde su móvil.

3.2.1.3 Mecánicas

Todo el diseño del juego se construye sobre una sola mecánica: el *parry.* El *parry*, o bloqueo, es un termino utilizado en videojuegos para referirse a bloquear un ataque en el momento justo, siendo capaz de devolverlo o contraatacar a la persona que te ha atacado. En este juego **no** hay una mecánica de *parry*, pero la idea de como se golpea la pelota vino de pensar como se podría implementar un *parry* en juego de este estilo.

El juego no tiene un botón dedicado a golpear la pelota (de hecho, no tiene ningún botón salvo el *joystick* de movimiento), si no que este golpe viene del hecho de no pulsar ningún botón. La manera de jugar es estar todo el rato moviéndose por la pantalla y en el momento en que la pelota se acerca a tu personaje, dejar de moverte, lo que hará que el personaje batee lo que tenga en frente. Al igual que en un *parry*, este viene de una acción defensiva en vez de ofensiva y en los dos hay que esperar al momento justo para realizarla y devuelve el ataque a quien lo ha lanzado.

Aunque la mecánica de golpear una pelota es extremadamente común, esta manera de ejecutarla no es especialmente ortodoxa y puede ser incomoda de llevar a cabo como jugador, así que necesitara apoyarse de otras mecánicas para poder funcionar con todo su potencial.

Necesitamos que el jugador tenga la necesidad de mantenerse en movimiento en todo momento para que si el acto de pararse en seco tenga un peso real. Para ello, antes de poder golpear la pelota, el jugador tiene que haber cargado un medidor que se carga mientras se mueve. En el momento en el que deja de moverse el medidor se vacía por completo, y solo se batea si el medidor estaba cargado.

Para añadir una relación de riesgo/recompensa a esta mecánica, el medidor tiene tres niveles posibles de carga, cada una tardando más que el anterior en cargarse pero golpeando la pelota más fuerte, por lo que el jugador siempre tiene un incentivo para intentar apurar el medidor lo máximo posible antes de golpear la pelota.

Cuando el jugador batea, el bate hace un arco en frente de el y golpea todo el área que ha recorrido. Esta está marcada en todo momento con un indicador cónico a los pies del personaje. El punto de este área donde se encuentra la pelota es importante, ya que si se encuentra en el centro al ser golpeada saldrá disparada en la dirección exacta a la que se esta apuntando, pero si se encuentra en la periferia el tiro sera mucho menos preciso.

Este grupo de mecánicas es muy simple y algo limitado, pero teniendo en cuenta el tiempo y la escala de este proyecto son las adecuadas para que ninguna de ellas quede descuidada y puedan implementarse sin hacer ningún tipo de concesión.

3.2.1.4 Controles

Como se ha explicado al inicio, el juego se debe controlar con solo una mano, por lo que los controles tienen que ser muy sencillos.

Para moverse se usara un *joystick* virtual situado en el centro de la parte inferior de la pantalla que es capaz de seguir tu dedo por toda la parte inferior de la pantalla del móvil, por lo que toda esta área ocupada por él. Esto nos deja sin espacio para ningún otro botón, ya que la mitad superior esta demasiado lejos como para poder llegar a ella con el dedo cómodamente.

Por eso para controlar hacia donde mirara nuestro personaje, y por tanto hacia donde bateara, utilizamos el giroscopio del móvil para apuntar. Al rotar el móvil en horizontal la dirección a la que mira el personaje rota de la misma manera.

3.2.1.5 Arte

El juego debe de tener una estética limpia para se pueda distinguir de un solo vistazo lo que esta pasando en pantalla. Esta también debe extenderse a los menús.

En aras de la simplicidad, el juego no perseguirá el realismo sino que tendrá buscara ser estilizado y simple. Tiene que tener un aire futurista pero manteniéndose simple.

Ya que el juego es una competición de un jugador contra otro, se le asignara un color a cada jugador (rojo y azul) y se utilizara esta dicotomía de colores como paleta de color para los menús.

3.2.2 Programación

3.2.2.1 Mecánicas del jugador

La escena *player* que representa al jugador está dividida en varios nodos que implementan diferentes mecánicas y funcionalidades, con un *script* para coordinar estas funcionalidades en el nodo raíz de la escena. La mecánica más básica de todas sería el movimiento del personaje, y para implementar se ha utilizado el sistema de *Inputs* de *Godot.* Este permite asignar crear variables y asignarles uno o más botones, lo cual es muy útil para programar diferentes métodos de control, como teclado y mando. En nuestro caso, estas variables están conectadas al *joystick* táctil y permiten utilizarlo como si fuera cualquier mando.

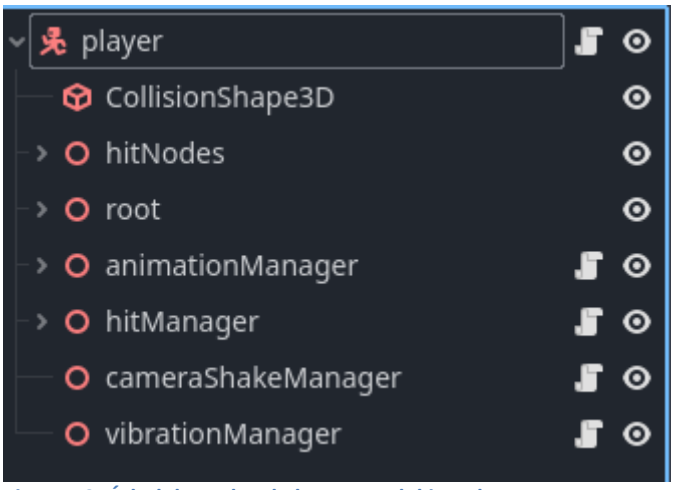

Figura 40: Árbol de nodos de la escena del jugador

En el *script* del jugador obtenemos los valores de estas variables y la multiplicamos por la base[11] del jugador para obtener la dirección en la que este se debe de mover. Normalizamos el vector que nos indica la dirección para que no influya en la velocidad para más tarde multiplicarlo por el valor de velocidad deseado. Como no queremos que el jugador pueda moverse en el eje Y multiplicamos el valor Y del vector por cero. Gracias a que el jugador es un nodo del tipo *CharacterBody3D* podemos utilizar la función *move_and_slide*, pasarle el valor que acabamos de calcular y dejar que el motor se encargue del resto. Para facilitar el poder defender la portería, aumentamos ligeramente la velocidad del jugador si está caminando en dirección a su propia portería.

11 Una base en álgebra lineal se caracteriza como un conjunto de elementos linealmente independientes entre sí que constituyen un sistema generador del espacio vectorial al que pertenecen [24] .

```
var input_dir := Input.get_vector( "move_right", "move_left", "move_down" , "move_up")
direction = (transform.basis * Vector3(input_dir.x, 0, input_dir.y))

if direction:
	var speed = SPEED
	if (team.name == 'Team A' and direction.z > 0) or (team.name == 'Team B' and direction.z < 0):
		speed *= backwards_speed_boost

	velocity.x = direction.normalized().x * speed
	velocity.z = direction.normalized().z * speed

	root.look_at(global_transform.origin - direction, Vector3.UP)
	animation_manager.moving()
	hit_manager.charging = true;
else:
	velocity.x = move_toward(velocity.x, 0, SPEED)
	velocity.z = move_toward(velocity.z, 0, SPEED)

move_and_slide()
```

Para saber cuando el jugador debería estar cargando su medidor detectamos si el *joystick* virtual está siendo tocado y le indicamos al nodo *hitManager,* el encargado de gestionar este medidor, que inicie la carga. En caso de que se deje de pulsar se comprueba si se había llegado a cargar y en caso positivo se le envía una señal al jugador para que empiece la animación de golpear la pelota. Antes de decidir si se ha dejado de pulsar el *joystick* se dejan unos pocos fotogramas de margen para evitar que el usuario pueda parar la carga por accidente.

```
if movement:
	_last_movement = 0
	_calc_movement(delta)
return

if _last_movement > time_to_stop_charge:
	hit_manager.charging = false;
	_last_movement = 0

_last_movement += 1
```

En un inicio, el indicar de carga tenía un solo nivel, en el que se llegaba a la carga o no, pero después se añadieron dos niveles más. Cuando el indicador se acaba de cargar, vuelve a empezar la carga rellenándose de color azul, y si esta se acaba se repite lo mismo en color rojo, cada una tardando más en cargarse y golpeando la pelota más fuerte que el anterior, creando una relación de riesgo y recompensa en la que se debe elegir con cuidado cuando se quiere golpear.

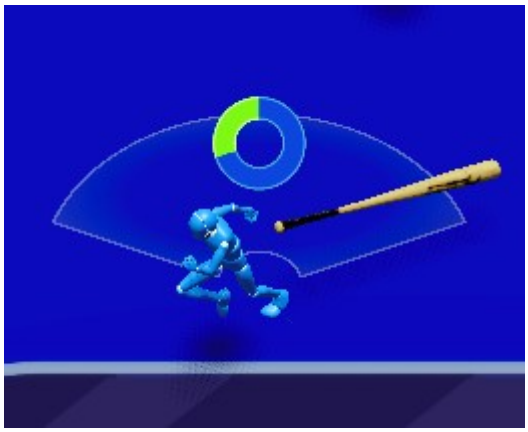

Figura 41: Indicador de carga pasando del primer al segundo nivel.

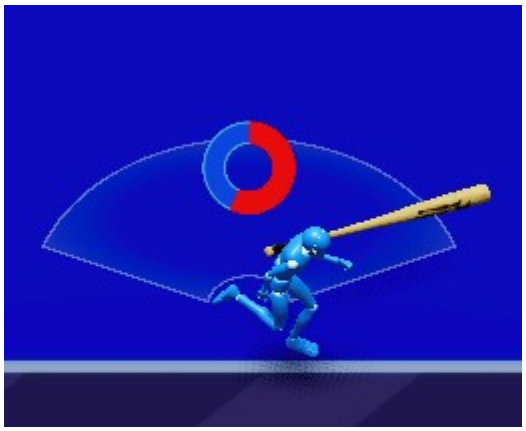

Figura 42: Indicador de carga pasando del segundo al tercer nivel.

Para saber cuando estos medidores han terminado de cargarse no contamos unidades de tiempo, sino que comprobamos directamente el porcentaje de carga del medidor durante cada fotograma. Para cargarlos, le sumamos durante cada fotograma la misma cantidad al valor del medidor, que tiene configurado un valor máximo de carga. Como a todos los medidores les sumamos la misma cantidad cada vez, mientras más carga total tenga más tiempo tardara en cargarse. Cuando el valor total supera o iguala al valor máximo marcamos que se ha alcanzado ese nivel de carga y iniciamos el siguiente marcador.

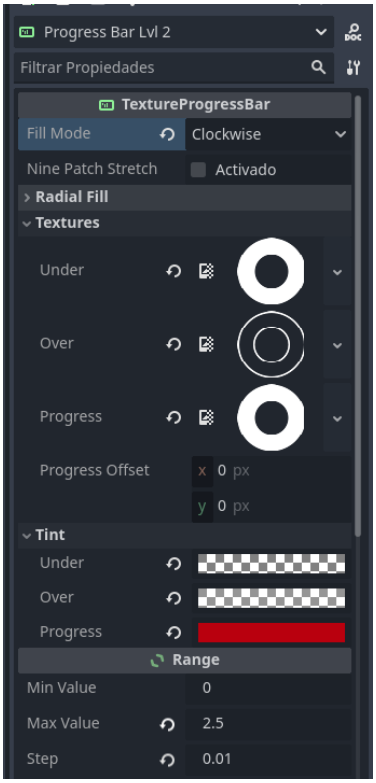

Figura 43: Vista del inspector de uno de los nodos *TextureProgressBar* que indican el nivel de carga.

```
1  func _physics_process(_delta):
2          if charging:
3                  texture_progress_bar_0.value += 0.01
4                  if texture_progress_bar_0.value >=
texture_progress_bar_0.max_value:
5                          charge_level = 1
6                          texture_progress_bar_1.value += 0.01
7                          if texture_progress_bar_1.value >=
texture_progress_bar_1.max_value:
8                                  charge_level = 2
9                                  texture_progress_bar_2.value +=
0.01
10                                 if texture_progress_bar_2.value
>= texture_progress_bar_2.max_value:
11                                         charge_level = 3
```

Para añadirle más peso al golpear la pelota, la cámara como el teléfono vibran en el momento del impacto, y mientras mayor nivel de carga tuviera el golpe, más vibran. La vibración de la cámara se hace utilizando el sistema de *tweens* y una textura de ruido para aleatorizar la vibración. Para hacer vibrar el teléfono se puede utilizar la función *vibrate_handheld* de *Godot* siempre que se activen los permisos necesarios a la hora de exportar el juego.

```
1 func _on_hit_area_body_entered(body):
2       if body is Ball:
3               var hit_direction = hit_nodes.global_transform.basis * Vector3.FORWARD
4               camera_shaker.shake_camera(hit_manager.charge_level)
5               vibrator.vibrate(hit_manager.charge_level)
6               if is_multiplayer:
7                       body.kick.rpc(-1 * hit_direction.normalized()*hit_manager.get_kick_force())
8               else:
9                       body.kick(-1 * hit_direction.normalized()*hit_manager.get_kick_force())
```

Durante el prototipado, la pelota se golpeaba en la misma dirección a la que estuviera mirando el jugador, pero eso resulto ser un sistema algo incómodo y confuso, así que se decidió que la dirección de movimiento y de golpeo tenían que ser independientes. Para ello, y teniendo en cuenta que no se podían añadir más botones a la interfaz sin romper el diseño de jugar con una sola mano, se optó por utilizar el giroscopio del móvil para controlar la dirección. Por suerte la función *get_gravity* devuelve los datos del giroscopio teniendo en cuenta la aceleración gravedad de la tierra para que no influya en las lecturas. Estas se utilizan para rotar *hitNodes,* un conjunto de nodos que contiene tanto el *Area3D* que detecta la colisión con la pelota como las indicaciones visuales para que el usuario sepa hacia donde está apuntando. Este ultimo se encuentra normalmente desactivado y solo se activa durante la animación de golpear la pelota.

Al golpear la pelota se tiene en cuenta su velocidad para mantener parte de esta en el nuevo impulso, así que además de añadirle el impulso del golpe se le suma un cuarto de la velocidad que ya llevaba la pelota pero en sentido contrario. Antes de aplicar el nuevo impulso se frena completamente la pelota para asurarse de que al golpearla como mínimo la paras en seco, ya que si dejamos a la física actuar por su cuenta en muchas ocasiones la pelota lleva tal impulso que no es posible pararla con un golpe cargado de primer nivel y puede legar a ser muy frustrante para el jugador.

```
1 var velocity = linear_velocity / 4
2 linear_velocity = Vector3(0,0,0)
3 apply_impulse(force - velocity)
```

3.2.2.2 Mecánicas del rival

La escena del rival controlado por el juego hereda de la escena del jugador, así que gran parte de lo explicado en el apartado anterior sigue siendo válido para este. La diferencia más importante es que el *script* que une todas las partes es diferente.

Para empezar, toda la mecánica de movimiento comparte el mismo código que el jugador con la diferencia de que para obtener la dirección a la que hay que moverse se consulta a un nodo

N*avigationAgent3D* para que este indica la siguiente posición a la que hay que moverse. Este nodo que no existía en la escena del jugador se utiliza para integrar el *pathfinding*[12] en el movimiento del personaje. El funcionamiento del nodo es muy simple, solamente hay que indicarle las coordenadas del punto al que se quiere ir, y se pueden configurar varios parámetros como que algoritmo de *pathfinding* se usara, el tamaño máximo del camino calculado o a cuantos metros del punto deseado se quiere acabar. Con esto configurado, basta con llamar a la función *get_next_path_position* del nodo durante cada fotograma, que devolverá la siguiente posición a la que debería moverse el personaje si quiere seguir el camino y llegar a su destino. Con esta posición, y la posición actual del personaje, podemos calcular hacia qué dirección se debe mover y así podemos reutilizar todo el sistema de movimiento del jugador.

```
1 var current_location = global_transform.origin
2 var next_location = nav_agent.get_next_path_position()
3 var direction = (next_location - current_location).normalized()
```

Para elegir en que dirección debe estar apuntando, siempre tiene en cuenta su posición relativa con la posición de la portería rival. Como seria muy injusto que siempre estuviese apuntando en la dirección de la portería, se apunta en un angulo en diagonal en la dirección de la portería, para asegurarse de que la pelota ira hacia el lado correcto pero tendrá varios rebotes antes de llegar la portería, dándole así más tiempo de respuesta al jugador .

La verdadera diferencia entre el jugador y el rival se presenta cuando hay que decidir hacia donde se moverá el rival, lo que sería programar su inteligencia artificial y para ello, se tomó la aproximación más clásica: una máquina de estados. Esta máquina tiene dos partes bien diferenciadas, lo que sería la propia máquina en sí, y los diferentes estados. Para crear este tipo de arquitectura en *Godot*, lo más fácil es hacer la máquina que gestionara los estados el nodo padre, y cada uno de estos estados como un nodo hijos.

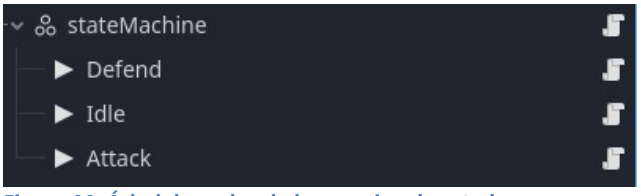

Figura 44: Árbol de nodos de la maquina de estados

Esta máquina guarda todos los estados en un diccionario durante el *_ready*, hace que el estado activo se ejecute cada fotograma durante el *_process* y se encarga de gestionar el cambio entre estados. Para saber cuando ha de cambiar de estado, está conectado a una señal que emiten los estados cuando quieren hacer el cambio. Cada uno de estos estados hereda de un nodo llamado *EnemyState* que implementa todas las funciones necesarias de los estados. Cada estado ejecuta código tanto al entrar como al salir del estado, y durante cada fotograma que está activo, y tiene que indicar cuando quiere emitir la señal de cambio de estado y hacia cuál quiere cambiar. Con este sistema montado, la inteligencia artificial del rival tiene tres estados:

12 Se denomina *pathfinding* en inglés, al trazado por una aplicación de computadora, del camino más corto entre dos puntos [25] .

- Idle: Este estado existe para hacer preparaciones para el resto de estados mientras carga la partida. En el momento en el que empieza la partida, se cambia al siguiente estado.

- *Defend*: Mientras la pelota no este en su lado del campo se mantendrá este estado. Durante este, el rival se moverá de manera aleatoria en vertical, pero seguirá la posición horizontal de la pelota en todo momento para estar listo para interceptarla en caso de que entre en su campo. Estará moviéndose en todo momento para tener siempre que sea posible un golpe cargado en caso de tener que golpear la pelota. En el momento en el que se detecte que la pelota entra en su campo, se pasara el estado *Attack.*

- *Attack*: en este estado el personaje se moverá hasta la posición de la pelota y si detecta que la pelota está a una distancia a la que la puede golpear y tiene un golpe cargado, bateara en dirección a la portería. Cuando la pelota deje su zona del campo, volverá al modo *Defend.* En caso de querer golpear la pelota, pero no tener suficiente carga, se pasara al estado *Charge*.

- *Charge:* este estado se mantendrá siempre lo más cerca posible de la pelota mientras se mueve, y en el momento en el que la carga haya llegado al primer nivel, volverá al estado *Attack*

Este comportamiento, pero, depende de una pieza fundamental para funcionar. Si se hiciera esto siguiendo la posición de la pelota el rival sería demasiado lento como para poder actuar, así que tiene que ser capaz de anticiparse a los movimientos de la pelota. Para ello, se calcula en todo momento la futura posición de la pelota, y se hacen todas las comprobaciones teniendo en cuenta este valor, en vez de la posición actual. Para ello, teniendo en cuenta que se conoce la posición y velocidad de la pelota en todo momento, se emplean las ecuaciones del movimiento rectilíneo uniforme para obtener la posición de la pelota cien milisegundos en el futuro.

```
1 func _process(_delta):
2         var future_position = transform.origin + linear_velocity * future_position_time
3         future_position_node.transform.origin = future_position
```

3.2.2.2.1 Problemas

Como este rival reutiliza las mismas mecánicas que el jugador es muy importante controlar cuando se mueve y cuando se para, ya que eso sera lo que haga que su indicador de carga aumente y que batee.

Ha habido muchos problemas para que el enemigo se moviese continuamente y no se alejase demasiado del área en la que iba a estar la pelota. Para regular esto ha habido que ajustar mucho los limites de la generación aleatoria de posiciones a las que se podía mover, todo a base de prueba y error, hasta encontrar el valor exacto que hace que nunca se dejara de mover pero que tampoco se moviese demasiado.

Aunque utiliza una maquina de estados muy simple, la transición entre estos a dado muchos problemas. El principal error ha sido que se quedara estancado entre dos estados, o transicionando en bucle entre ellos, ya que hay situaciones muy especificas que pueden llevar

el sistema al limite. También era común que al pasar entre estados el personaje se quedase quieto durante unos momentos, tiempo suficiente para que hiciera un golpe y perdiese toda su carga.

La solución a estos problemas ha sido controlar más cuando y como se cambia entre estados, añadiendo más variables para controlar las posibles variaciones y asegurándose de que al entrar en un estado nueva se inicializaran ciertos valores importantes, para no encontrarse más tarde información inesperada en estas variables.

3.2.2.3 Gestión de la partida

Para gestionar el inicio y el final de la partida se crean dos nodos, *Game Manager* para gestionar el estado de la partida y *Score Manager* para gestionar la puntuación. Este último tiene conectadas dos señales, una de cada portería, que se emiten al detectar una colisión con la pelota y que utiliza para actualizar la puntuación de cada jugador y mostrarla por pantalla durante unos pocos segundos utilizando el nodo *Timer* para llevar la cuenta, además de emitir una señal para devolver la pelota a su posición inicial. Cuando detecta que la puntuación de alguno de los dos jugadores ha llegado al límite de la partida, indica que jugador ha ganado y emite una señal para acabar la partida.

```
1  if finished or (score_A < score_to_win and score_B <
score_to_win):
2          return
3  if score_A >= score_to_win:
4          label_score_A.text = "WINS"
5          label_score_B.text = "LOSE"
6  if score_B >= score_to_win:
7          label_score_A.text = "LOSE"
8          label_score_B.text = "WINS"
9  finished = true
10  score_timer.wait_time = end_time
11  _show_score()
```

```
1  func _hide_score():
2          UI_scores.visible = false
3          if finished:
4                  exit.emit()
```

El nodo *Game Manager* se encarga de detectar cuando ha cargado la partida para dejar de mostrar la pantalla de carga, mueve la pelota al punto indicado después de cada gol y maneja el menú la pausa del juego cuando se abre el menú. Por último, es el encargado de terminar la partida, tanto cuando alguno de los jugadores ha ganado como cuando uno sale de la partida desde el menú.

Hay un nodo más que se utiliza para la gestión, y que esta siempre cargado gracias al a*utoLoad.* Este nodo guarda dos valores, la velocidad a la que ira la pelota durante la partida, y cuantos puntos hay que tener para ganar. El *score manager* lee este valor durante el inicio de la partida, y la pelota lee el valor de velocidad y lo utiliza como un multiplicador cuando calcula su velocidad. Los valores de este nodo han sido configurados antes de empezar la partida.

3.2.2.4 Multijugador

3.2.2.4.1 Lobby

Para empezar a hablar del multijugador, primero hay que ir al menú principal del juego. Una de las limitaciones que tiene el sistema *peer to peer* es que antes de poder empezar a jugar se

tiene establecer una comunicación previa entre los jugadores, por lo que los juegos suelen emplear un sistema de *lobby* en el que juntar a los jugadores antes de que empiece la partida.

Eso mismo es lo que se implementa en este proyecto: primero se crea una clase *lobby*, que se encargara de iniciar las conexiones entre pares, mantenerlas activas y cortarlas cuando sea el momento, y estará activo en todo momento utilizando el *autoload* de Godot. En esta clase se utilizará tanto la API de bajo nivel como la de alto nivel.

El primer objetivo es que un usuario pueda crear una partida en la que haya un hueco para que otro usuario se una, y que cuando la conexión entre ambos se haya establecido correctamente se pueda iniciar la partida. Para facilitar la implementación, el jugador que cree la partida actuara tanto de servidor como de cliente, mientras que el que se una únicamente actuara como cliente. Tomamos esta decisión porque la API de alto nivel de *Godot* favorece arquitecturas de red en la que solo hay una autoridad y permite perfectamente que el mismo dispositivo actué como servidor y como cliente.

Este *script*, *lobby*, tiene que ser capaz de comunicarse con otros scripts para indicarles las conexiones entrantes, así que creamos tres señales para esto mismo.

```
1 signal player_connected(peer_id, player_info)
2 signal player_disconnected(peer_id)
3 signal server_disconnected
```

La primera de ellas se utiliza para indicar que se ha empezado la conexión con otro par, la segunda indica que esa conexión ha terminado, y la tercera que el par que actúa como servidor ha cortado la conexión. Para saber cuando se han de emitir estas señales, conectaremos señales de la API multijugador de alto nivel de *Godot* a diferentes funciones de este script.

Pero antes de continuar hay que crear un *array*[13] en la que se guardara la información de los usuarios conectados y hay que elegir el puerto por el que se enviaran los paquetes. Después de comprobar que no es un puerto que ya se esté utilizando por otros programas, se eligió el puerto 7000, por ninguna razón en particular. Con todo esto listo, se puede comenzar a programar el *lobby.*

En el *_ready* se conectan varias señales importantes de la API de *Godot,* a la que se accede utilizando la variable *multiplayer*.

```
1 func _ready():
2       multiplayer.peer_connected.connect(_on_player_connected)
3       multiplayer.peer_disconnected.connect(_on_player_disconnected)
4       multiplayer.connected_to_server.connect(_on_connected_ok)
5       multiplayer.connection_failed.connect(_on_connected_fail)
6       multiplayer.server_disconnected.connect(_on_server_disconnected)
```

La primera de estas funciones, *_on player_connected*, se llamará siempre que se conecte con otro par, ya sea este el servidor o no, y se encargará de ejecutar una función RPC configurada para ser ejecutada por todos los pares. Esta función guarda la información del par recién conectado y emite la señal *player_connected*.

13 En programación, se le denomina *array* a una zona de almacenamiento contiguo que contiene una serie de elementos del mismo tipo [29] .

La segunda función, *_on_player_disconnected*, se llama cuando se desconecta un par y simplemente elimina la información del par previamente conectado y emite la señal player_disconnected.

La tercera, *_on_connected_ok* se llama cuando un par se conecta con un par que está haciendo la función de servidor, y hace lo mismo que la primera función, pero sin llamar a ninguna función RPC, por lo que solo se ejecuta en el par que se ha conectado, no en el servidor.

La siguiente, *_on_connected_fail*, se ejecuta cuando la API detecta cualquier tipo de error de conexión, y se encarga de cortar completamente la conexión con el par. Esta función puede parecer redundante, pero ayuda a evitar problemas de conexión cortando el problema de raíz.

Por último, *_on server_disconnected* se llama cuando el el par que actúa de servidor corta la conexión, y se utiliza para limpiar la lista de jugadores conectados y emitir la señal *server_disconnected*.

3.2.2.4.2 Conexión entre jugadores

Con todas estas señales conectadas ya podemos responder correctamente a los diferentes eventos en la conexión, así que el siguiente paso es permitir que estas conexiones se puedan realizar, añadiendo funciones para crear y unirse a una partida. Para crear una partida, en la función *create_game* creamos una variable del tipo *EnetMultiplayerPeer*, que se utiliza para crear un par en una conexión. Indicaremos que este par actuara como servidor y le indicamos que puerto utilizara para la conexión, comprobamos si ha habido algún error durante la creación del par y en caso de haberse creado correctamente, indicamos a la API que queremos establecer una conexión con el par, nos guardamos sus datos y emitimos la señal *player_connected* ya que se esta actuando como jugador y como servidor al mismo tiempo.

El par servidor siempre tendrá un uno como número de identificación, así que podemos utilizarlo para saber en todo momento que par ha creado la partida. Cuando creamos la conexión con este par, le indicamos que utilice al algoritmo de compresión *COMPRESS_RANGE_CODER* para los paquetes de red, que están siendo enviados por UDP. Este algoritmo es muy eficiente en la compresión de paquetes pequeños, de menos de 4KB, así que parece adecuado para este proyecto en el que los paquetes tendrán mayoritariamente información sobre el movimiento de los personajes y la velocidad y posición de la pelota [30] .

La función *join_game* es muy similar, con la diferencia de que al crear el par le indicamos que sera un cliente y le pasamos una dirección IP a la que conectarse. Por defecto, esta dirección apunta al propio dispositivo que esta realizando la conexión, así que si no se inserta ninguna dirección al llamar a la función se intentara conectar a si mismo, lo que solo funcionara en caso de que haya dos instancias del juego abiertas en el mismo dispositivo y una de ellas haya creado una partida. La otra diferencia que tiene con la anterior función es que no emitimos la señal *player_connected* ni nos guardamos la información del par*,* ya que como hemos mapeado anteriormente las señales de la API a diferentes funciones, si la conexión con el servidor se ha realizado correctamente esta emitirá *connected_to_server* lo que hará otra función haga estas dos cosas.

3.2.2.4.3 Lista de partidas

Todo esto permite a dos jugadores establecer una conexión entre si y empezar a intercambiar paquetes, pero tiene un problema. En estos momentos la única manera de conectarte a un servidor es introducir la dirección IP del jugador que ha creado la partida, lo que, a nivel de experiencia de usuario, no es aceptable: Tiene que haber una manera de detectar partidas creadas dentro de tu red local para poder unirte a ellas fácilmente, así que para ello creamos la clase *Server Browser.* Esta clase extiende de un nodo de control, que se utiliza para crear y mostrar una interfaz de usuario, pero en este apartado solo se hablara de las partes relacionadas con el manejo de la conexión de red.

Este sistema funcionará de la siguiente manera: cuando un jugador haya creado una partida se empezarán a enviar paquetes en *broadcast,* un tipo de multidifusión de paquetes que permite enviarlos a todas las direcciones IP de la red. Estos paquetes contendrán el nombre del jugador que ha creado la partida para poder mostrarlo en la lista, y el número de jugadores de la partida para poder saber si está llena o no.

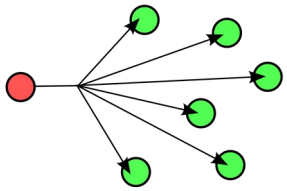

Figura 45: Diagrama ejemplificando una conexión multidifusión de tipo *broadcast*

Esta nueva clase empieza creando la señal *server_found*, con los argumentos *server_ip* y *player_name*, que se emitirá al detectar una partida creada. La primera variable de la clase será un *Timer* que indicara cada cuanto tiempo se enviaran los paquetes de red, inicialmente configurada para actuar una vez por segundo. Las siguientes son el puerto por el que se enviaran los paquetes en *broadcast* y el puerto que se utilizara para capturar estos paquetes, seguidos de la dirección de *broadcast* a la que se enviaran. Dependiendo de la arquitectura de la red en la que se esté trabajando se pueden utilizar diferentes direcciones, aunque 255.255.255.255 funcionara en cualquier red. En Europa las redes suelen venir configuradas de manera en que 192.168.1.255 funcionara de igual manera y nos ayudara a ahorrar algo de ancho de banda, aunque esto podría no funcionar en determinadas configuraciones.

También se utilizarán dos variables para guardar los datos de las partidas y sus IP respectivamente, y por último dos variables de tipo *PacketPeerUDP.* Estas son parte de la API multijugador de bajo nivel de *Godot*, y sirven para enviar y capturar paquetes UDP; una se llamará *listener* y la otra *broadcaster.* Para utilizarlas, tenemos que inicializarlas cuando queramos empezar a emitir o a capturar y configurarlas con los valores de las variables anteriormente mencionadas.

Cuando el temporizador emita su señal, se enviarán los paquetes con la información pertinente.

```
# Usamos \n para separar informacion
var data = room_info.name + "\n" + str(room_info.count)
var packet = data.to_ascii_buffer()
broadcaster.put_packet(packet)
```

En el *_process* de esta clase se comprueba si el *listener* ha capturado algún paquete, y si se da el caso se obtiene su información junto con la IP del remitente, se guarda en una lista y se emite la señal que hemos creado al inicio. Esta clase no se encarga de manejar la conexión, solo de obtener estas listas, que serán pasadas a la interfaz de usuario. En el momento en el que esta clase deja de estar en la escena, se apagan el *listener* y el *broadcaster.*

3.2.2.4.4 Sala de espera

Para utilizar la información recibida de esta clase para mostrar las partidas, seleccionar a la que unirse y pasar la información a la clase *Lobby* que se encargara de realizar la conexión se utilizara el Menú Multijugador , del cual se hablara en otro apartado.

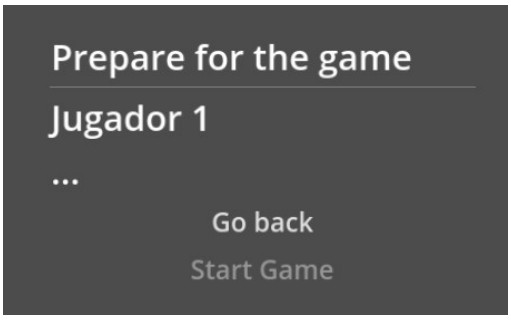

Figura 46: Lobby de una partida con un solo jugador

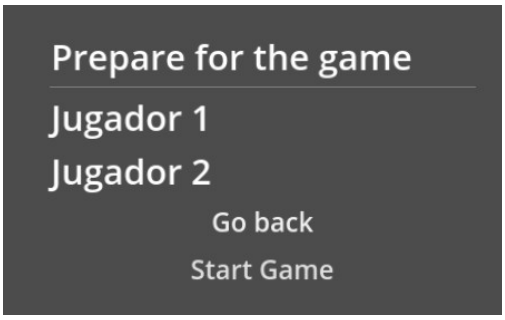

Figura 47: *Lobby* de una partida con dos jugadores conectados

Cuando un jugador crea una partida, en realidad este creando un *lobby* en el que espera a que otro jugador se una, y en el momento en que eso pase puede darle a un botón para empezar la partida. Cuando los dos jugadores están en el *lobby* la conexión entre ellos se ha establecido satisfactoriamente y el *lobby* ningún jugador más puede acceder a él. Con esta conexión establecida se puede empezar a trabajar en cargar la escena de la partida, instancia un personaje por cada jugador y sincronizar los movimientos que cada uno realiza en su móvil para que cada uno controle a un personaje y todos los movimientos estén sincronizados entre los dos.

3.2.2.4.5 Iniciar una partida

Para simplificar las explicaciones, a partir de ahora nos referiremos al jugador que crea la partida como servidor y al que se une como cliente. Cuando se pulsa el botón de iniciar partida, se hace una llamada RPC a una función de lo *lobby* que carga tanto en cliente como en el servidor la escena de la partida multijugador. La escena de la partida, que es donde ocurre el juego como tal, realmente está dividida en tres escenas: la primera, *game_base* tiene todo lo que se utilizara tanto en el modo de un jugador como en el multijugador, como la pelota, el *game manager*, todo el escenario, las luces y la UI, mientras que hay otras dos escenas que heredan de esta, *game-sp* y *game-mp*. La primera añade los elementos únicos del modo un jugador, como el jugador rival controlado por IA, mientras que el segundo añade los del multijugador como la cámara para cada jugador o nodos para sincronización.

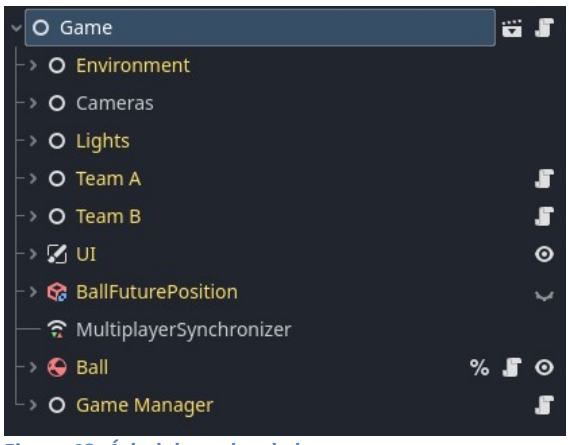

Figura 48: Árbol de nodos de la escena *game-mp*

Figura 49: Árbol de nodos de la escena *game-sp*

Como se puede ver en la Figura 48, el nodo raíz de *game-mp* tiene un *script*: este se llama *MultiplayerGame* y se encarga de asegurarse de que los dos jugadores están sincronizados antes de empezar la partida. En el *_ready* activa una pantalla de carga y le dice a la clase *lobby* que ya ha cargado la escena utilizando una llamada RPC, y cuando *lobby* detecta que los dos jugadores ya la han cargado, le dice a *MultiplayerGame* que puede empezar a sincronizar a los jugadores. Para ello, llama a la función *_spawn_players*, que instancia a un personaje para jugador y dependiendo de la ID de red que tenga cada uno (recordemos que el servidor siempre tiene ID=1) les asigna una posición, una cámara y un equipo. Esta función se llama utilizando RPC configurado para que solo el servidor pueda llamarla, por lo que se ejecutara tanto en cliente como en servidor, pero solo cuando el servidor lo decida, evitando así que la llame el cliente y el servidor y se ejecuta más veces de la cuenta por error. Como se está llamando vía RPC, se ejecutara de manera sincronizada en ambos y permitirá que la partida comience completamente sincronizada entre ambos. Cuando se han instanciado los dos personajes, se emite una señal conectada al *game manager* para ocultar la pantalla de carga y marcar el inicio de la partida.

En este momento, cuando se inicia la partida ambos jugadores ven lo mismo pero espejado en vertical, ya que en la escena hay dos cámaras, una en cada extremo del campo, y se las asignamos a cada jugador cuando instanciamos a su personaje. El problema viene en que, cuando el cliente intenta mover a su personaje, este se queda quieto, mientras que sí lo hace el servidor mueve a los dos personajes, aunque el cliente no lo verá en su pantalla. Esto es una mezcla de dos problemas: que el *input* de los dos jugadores aún no está separado, y que el movimiento de los personajes (y de la pelota) no está sincronizado. Ambos podemos solucionarlos utilizando un nodo *MultiplayerSynchronizer,* del que se habló brevemente en la introducción a *Godot.*

3.2.2.4.6 Sincronización entre clientes

Para simplificar la separación entre el modo de un jugador y de multijugador, creamos una nueva escena heredada de *player*, que solo añadirá uno de estos nodos. Utilizaremos este nodo para sincronizar la posición del jugador, su ID, su rotación y varias variables relacionadas con las animaciones.

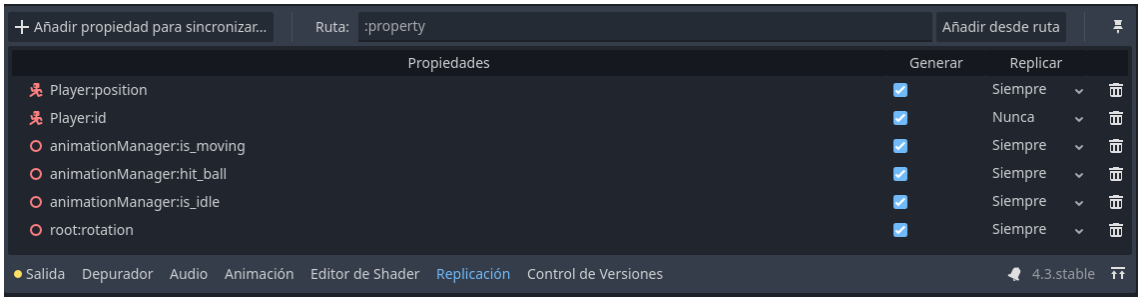

Figura 50: Cuando se utiliza un nodo *MultiplayerSynchronizer* se añade la pestaña Replicación a la interfaz, que nos deja seleccionar que propiedades de los nodos (variables) sincronizar y como hacerlo.

Esta escena mantendrá el mismo *script* principal que *player,* pero en este se comprobara si el nodo que hemos añadido existe, y en caso de que no lo haga sabremos que estamos en una partida de un solo jugador. Además, con este nodo podemos utilizar la función *get_multiplayer_authority* para obtener la ID del par que tiene la autoridad en esa partida; eso no significa obtener la ID del servidor, sino de quien debería estar controlando al personaje. Con esa información, podemos comprobar si esa ID coincide con la del par que está jugando, y así sabremos si debería o no controlar a este personaje.

```
1 func _is_this_not_multiplayer_authority() -> bool :
2         return is_multiplayer and mult_sync.get_multiplayer_authority() !=
multiplayer.get_unique_id()
```

Con esta información hacemos que el _process del *player* solo se ejecute para el jugador que debería estar controlandolo, haciendo que cada jugador mueva solo a un personaje. Como los movimientos de los personajes están sincronizados gracias al nodo *MultiplayerSynchronizer,* los movimientos de un jugador se reflejaran en la pantalla del otro, y tendremos el movimiento de los jugadores funcionando correctamente y sincronizado entre ambos.

Ademas esta información también se utiliza para saber de que color renderizar a cada personaje. Si el personaje no es la autoridad de multijugador, se le sustituye el material por uno igual pero de color azul.

La única pieza que falta para poder jugar una partida en multijugador es la pelota. Es de suma importancia que la pelota esté perfectamente sincronizada entre los jugadores, ya que al ser un objeto movido enteramente por físicas, la más ligera diferencia podría hacer que salga disparada en una trayectoria completamente diferente que haría imposible volver a sincronizarla. Durante el desarrollo, su correcta sincronización ha dado muchos problemas, pero al final se ha conseguido que funcionara sin errores combinando los dos siguientes métodos: primero, la función de golpear la pelota, en la que se le aplica la fuerza, se llama utilizando RPC, así el momento exacto en el que se aplica la fuerza es igual para ambos jugadores, y después se sincroniza la velocidad lineal, velocidad angular y *last_hit*, una variable que indica cuantos fotogramas han pasado desde que la pelota fue golpeada por última vez.

La sincronización de la pelota se realiza con un nodo de sincronización en la escena *multiplayer-mp*, igual que el que había en *player.* Esa última variable, *last_hit*, se utiliza para evitar que por culpa de la latencia se golpee la pelota dos veces seguidas, ya que se comprueba que hayan pasado al menos diez fotogramas desde la última vez que fue golpeada para asignarle un nuevo impulso. Con este sistema la trayectoria de la pelota se mantiene igual para los dos jugadores, no se producen situaciones en la que un jugador parece haber

golpeado la pelota sin haberle dado en realidad o al revés, y aun con cierta cantidad de latencia el movimiento de la pelota se ve fluido, ya que solo estamos sincronizando la velocidad mientras que dejamos que el sistema de físicas calcule el movimiento, por lo que los cálculos se realizaran localmente en cada dispositivo.

El toque final para permitir que se pueda jugar es ser capaz de marcar goles con pelota, y para ello se cambió la función que actualiza la puntuación de los jugadores para poder llamarse con RPC y que se ejecute tanto en cliente como en servidor. Como esta función se encarga de actualizar la puntuación, mostrarla por pantalla y emitir una señal para devolver la pelota a su posición inicial, tenemos todas las funcionalides necesarias ya sincronizadas. Para acabar, se cambia la función de terminar la partida para que desconecte a todos los pares al salir de la partida.

3.2.2.5 Interfaz de usuario

A nivel de programación la UI en Godot esta diseñada para trabajar con un modelo MVC (*Model-View-Controller)*, en el que un nodo se encarga de presentar la vista, otro de detectar la interacción y otro de implementar la funcionalidad del botón.

En este proyecto se ha implementado de la siguiente manera: cuando se crea un botón, este tiene una señal llamada *pressed* que se emite cuando se pulsa el botón. Esta señal se conecta a un nodo dedicado a detectar las pulsaciones de los botones de toda la escena, y este nodo llama a otros nodos para que lleven a cabo las funcionalidades correspondientes al botón. Estas funcionalidades están separadas en dos nodos, uno que se encarga de las animaciones de los diferentes botones, como por ejemplo esconder y enseñar diferentes menús, y otro que se encarga del resto de funcionalidades, principalmente cargar nuevas escenas o cambiar las preferencias del usuario.

Hay algunos botones con un funcionamiento ligeramente diferente, como los botones de conmutación, que funcionan con dos estados: encendido y apagado. Para estos botones se utiliza la señal *toggled*, que incluye el parámetro *toggled_on* que indica el estado del botón. Ademas, para casos en los que hay un grupo de botones de este tipo, pero solo uno de ellos debe estar encendido al mismo tiempo, se pueden crear un grupo de botones, que implementa automáticamente esta funcionalidad para todos los botones dentro del grupo.

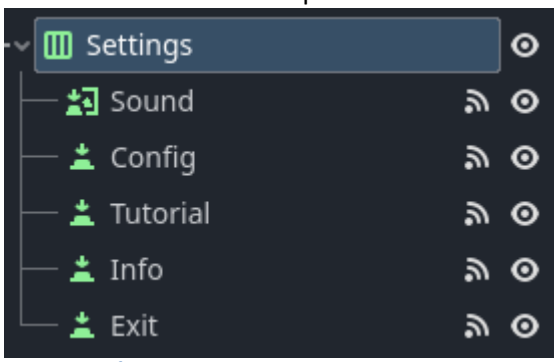

Figura 51: Árbol de nodos del menú de ajustes. Todos los botones tienen el icono que indica que una de sus señales ha sido conectada a un nodo.

Como el nivel de funcionalidad de la UI en este proyecto es bastante sencillo, con tres nodos por escena ha sido suficiente implementar y manejar todas las funcionalidades.

En general, Godot ofrece muchas facilidades para la creación de UI, ya que tiene muchos nodos especializados en las diferentes partes de la interfaz, tanto a nivel de gestión (indicar como se mostraran y se organizaran las diferentes partes de esta) como a nivel de funcionalidad (diferentes tipos de botones y *sliders* con muchas opciones de configuración), y el sistema de señales hace que interactuar con estos nodos sea cómodo e intuitivo. Todo esto ha hecho que el proceso de programar la interfaz haya resultado mucho más cómodo de lo que había imaginado.

3.2.2.5.1 Menú Multijugador

La única parte de la UI que ha resultado difícil de implementar ha sido la referente al multijugador, ya que hay que sincronizar todas las funciones de búsqueda de pares y conexión entre ellos con la interfaz para indicar al jugador en tiempo real el estado de la conexión y que este fuera capaz de conectarse y desconectarse a voluntad sin causar problemas de sincronización entre los pares.

Además, para la parte del buscador de partidas, estas tienen que ir apareciendo en la lista conforme se van encontrando, así que se tienen que instanciar de manera dinámica vía código. Para ello, se opto por crear un nodo para cada partida, que contendrá un *script* y un botón. El script esta configurado con una función que acepta como parámetros la *IP* de la partida y el nombre del jugador que la crea; el nombre se utilizara para mostrarlo como texto en el botón, mientras que la *IP* se guarda para utilizarse en la señal *server_pressed*, que se lanzara cuando se pulse el botón. Este nodo se guarda en memoria como un recurso y puede instanciarse dentro de cualquier otra escena cuando sea necesario, como hacemos cada vez que se encuentra una partida.

```
1 func _on_server_browser_server_found(server_ip: String, player_name: String) -> void:
2         var button_server: ServerButton = server_button.instantiate()
3         button_server.init(server_ip, player_name)
4         button_server.server_pressed.connect(_connect_to_server)
5         menu_servers.add_child(button_server)
```

Para que todos estos nodos se coloquen correctamente de manera dinámica sin tener que programar a mano sus posiciones los instanciamos dentro de un nodo de control del tipo *Container*, que se encarga de decidir la posición de los nodos que tiene dentro en base a ciertos parámetros. Para este caso, se utiliza un *VBoxContainer*, que coloca los nodos en vertical uno debajo de otro respetando los margenes y distancia que se configuren.

3.2.2.6 Sonido

Para manejar todo el sonido del juego Godot cuenta con un sistema de buses de audio con el que se puede separar todo el sonido en diferentes canales para poder añadirles efectos o cambiar sus volúmenes de manera independiente.

Figura 52: Buses de audio del proyecto

3.2.2.6.1 Música

El menú principal del juego esta conformado por dos escenas distintas, ya que toda la parte correspondiente al multijugador esta en su propia escena, por lo que para poder navegar por el menú sin que la música se cortase o volviese a empezar al cambiar entre escenas se ha creado un script para manejar toda la música del juego y se ha añadido al a*utoload* para que siempre esta cargado en el juego.

Este script extiende de la clase *AudioStreamPlayer,* que sirve para reproducir archivos de audio, y se encarga de reproducir la música del menú cuando nos encontramos en este, y la de la partida cuando esta empieza, independientemente de si estamos en un jugador o en multijugador. El *script* tiene dos funciones, una para cuando se entra a una partida y otra para cuando se sale de una partida, y se encargan de cambiar a la canción correspondiente y de conectar las señales de ciertos nodos a una de estas dos funciones, y esta configurado para emitir sonido por el bus de música.

En un inicio, el nodo raíz del menú principal tiene la señal *on_tree_exited*, que se llama cuando este nodo sale del árbol (por ejemplo, cuando se cambia de escena) conectado a la función *game_entered* del script, por lo que cuando se cambia de escena se llama a esta función, se comprueba si estamos en la escena de la partida, y en caso afirmativo se cambia la canción. Pero cuando volvemos a la escena del menú principal, hay que volver a conectar la señal al script, ya que se ha desconectado durante el cambio de escena.

Como las funciones del *script* se llaman en cuanto el nodo abandona el árbol de la escena, muchas veces la siguiente escena aun no ha cargado para cuando se realiza la llamada, lo que hace que no podamos comprobar en que escena nos encontramos. Para ello, a estos funciones se les instruye para esperar a que se haya terminado de procesar el fotograma actual antes de continuar con su ejecución con la siguiente linea:

```
await get_tree().process_frame
```

3.2.2.6.2 Efectos de sonido

Los efectos de sonido de este proyecto se dividen en dos partes, los menús y la partida.

Para los menús se busca que cada vez que se pulse un botón se reproduzca un sonido, para dar una sensación más táctil y dinámica a los menús, pero los botones de Godot no tienen ninguna opción para reproducir sonido, por lo que ha habido que implementarla. La manera más sencilla de hacer esto era conectar la señal que se emite al pulsar un botón con una función que reproduzca un sonido, pero conectar todos los botones a mano no era ni practico ni posible, ya que hay botones que se generan dinamicamente.

Para solucionar esto, se ha creado un *script* que se carga en el a*utoload* y que cada vez que se añade un nodo nuevo al árbol de la escena comprueba si este es un botón, y en caso positivo conecta la señal a su función *play_pressed* que se encarga de reproducir el sonido. Esta función además se encarga de variar de manera aleatoria el *pitch* del sonido para ayudar a que suene menos repetitivo.

```
playback.play_stream(sound, 0, 0, randf_range(0.8, 1.2))
```

Para los efectos de sonido que ocurren durante la partida, tenemos varios nodos encargados de reproducir los diferentes sonidos. Para el sonido de bateo, se ha creado un nodo dentro del jugador con una función llamada *play_bat_swing_sound* a la que le entra un parámetro que indica el nivel de carga del golpe. Cada nivel de carga tiene asignado un sonido diferente, y ademas el *pitch* de estos también cambia de manera aleatoria.

Esta función esta configurada para llamarse como *rpc* y que se ejecute en todos los pares, para que los dos jugadores escuchen el sonido cuando se juega en multijugador. La llamada a la función se hace al mismo tiempo que se llama a la función de reproducir la animación de bateo.

La pelota también hace un sonido al ser golpeada y las porterías cuando se marca un gol: esta funcionalidad se ha programado exactamente igual que el sonido de bateo del jugador, pero cada una en su nodo correspondiente. Todos estos sonidos se han configurado para reproducirse en el bus de efectos.

3.2.2.7 Animaciones

3.2.2.7.1 Personajes

Para manejar las animaciones, el jugador tiene un nodo llamado *animationManager* con un script que hace de intermediario entre el *script* del jugador y el *AnimationTree* que contiene las animaciones y gestiona las transiciones entre estas.

El *animationTree* tiene tres estados, cada uno con una animación asociada: *idle*, para cuando el personaje se encuentra quieto, *run* para cuando esta corriendo y *bat* para cuando esta bateando. Cada uno de estos estados constituye un nodo, que se conectan entre si por lineas que indican la transición entre los diferentes estados. Cada una de estas transiciones tiene diferentes configuraciones que alteran su comportamiento de dos maneras: el *switch mode* le indica si la transición debe ser instantánea o si debe esperarse a que la animación que se esta reproduciendo acabe antes de realizar la transición, y *condition* es una o varias variables de tipo *boolean* que indicaran si se debe realizar o no la transición.

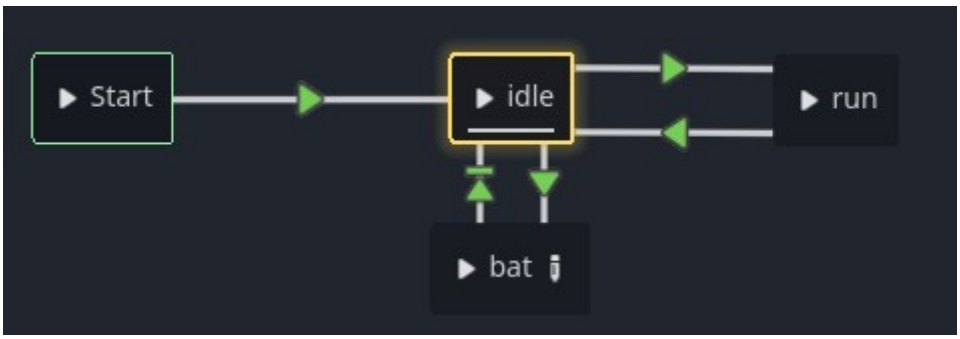

Figura 53: AnimationTree con todos los estados y transiciones del jugador.

Con esto se puede indicar todas las animaciones que realizara el personaje, cuando tiene que realizarse cada una y como tiene que ser la transición entre estas, pero utilizando solo los nodos básicos este sistema queda bastante limitado. Tanto el estado *idle* como *run* utilizan un nodo de tipo *AnimationNodeAnimation*, que se limita a reproducir una animación, mientras que el estado *bat* es un nodo del tipo *AnimationNodeBlendTree*: este nos permite no solo editar propiedades de la animación que queremos reproducir, como su velocidad, si no ademas interpolar de manera lineal entre dos animaciones para crear una animación a medio camino entre las dos y utilizarla durante la transición entre ambas, para hacer que el cambio entre ellas sea mucho más suave y realista.

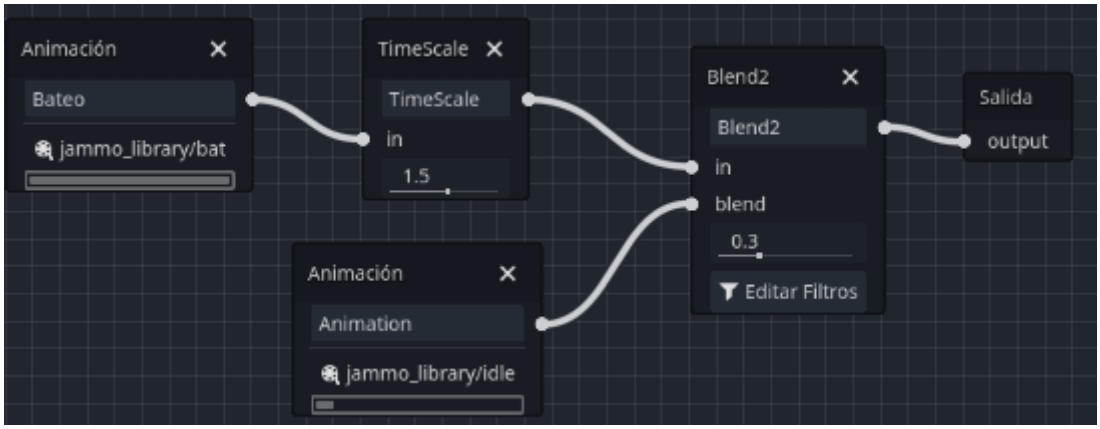

Figura 54: Interior del nodo *bat* del *AnimationTree* del jugador

Con el *AnimationTree* correctamente configurado solo falta indicarle los valores de las *conditions* para que sepa como navegar entre los diferentes estados. Estos valores los manejara el nodo *animationManager*, que tendrá una variable por cada *condition*, y se encargara de pasar estas variables al *AnimationTree* cada fotograma. Además, se asegurara de que estas no se contradigan entre si y de comprobar cuando se esta reproduciendo cada animación.

Como se ha aclarado antes, este nodo solo sirve para hacer de puente entre las animaciones y el *script* de comportamiento del jugador, así que sera trabajo de este ultimo indicar cuando se debe reproducir cada animación. Como todo el comportamiento ya esta definido añadir las animaciones es sencillo, simplemente se añade una llamada a la función correspondiente del *animationManager* durante los momentos correctos: cuando se esta procesando el movimiento del jugador se le indica que se active la animación de correr, cuando el jugador

esta quieto se indica que se vuelva al *idle*, y en caso de que el indicador de carga haya pasado el primer nivel, se indica que se reproduzca la animación de golpeo.

Como todo se maneja con tres variables, y cada una tiene una función asociada para activarla y desactivar el resto, manejar las animaciones es bastante trivial. El verdadero reto es configurar correctamente el *AnimationTree* y todos sus nodos para que no se produzcan cambios no deseados entre estados, que los cambios se hagan en el momento preciso y que las transiciones visuales entre las animaciones se produzcan de la manera más suave posible.

3.2.2.7.2 Menús

Para animar la UI de los menús el acercamiento fue completamente diferente, ya que en vez de crear animaciones a mano y manejar sus transiciones con un *AnimationTree* todas están animadas de manera programática. Hay muchas animaciones en los menús, y habría sido una inversión enorme de tiempo hacerlas todas a manos, por lo que en vez de eso se aprovecha el sistema de *tweens* de Godot, del que ya se ha hablado en este proyecto. Todas las escenas de menú tienen un nodo llamado *animation_manager* que se encarga de crear estos *tweens*.

En un inicio solo se muestran los botones del menú principal mientras que el resto están fuera del área visible de la pantalla, por lo que no se pueden ver. Cuando se pulsa un botón y se debe avanzar al siguiente menú, se llama a una función que mueve los botones correspondientes del menú fuera de la pantalla, y los del siguiente menú dentro. Para hacer que el movimiento se vea agradable a la vista utilizamos los *tweens*, que utilizan funciones matemáticas para interpolar el objeto deseado entre dos posiciones.

```
1 func _hide_main_screen():
2         var tween = create_tween()
3         tween.set_parallel(true)
4         tween.tween_property(start_screen_bottom, "position:x", 580,
1.0).set_trans(Tween.TRANS_EXPO).set_ease(Tween.EASE_IN_OUT)
5         tween.tween_property(lobby_menu_single, "position:x", 600,
1.0).set_trans(Tween.TRANS_CIRC).set_ease(Tween.EASE_IN_OUT)
6         tween.tween_property(lobby_menu_multi, "position:x", 600,
1.0).set_trans(Tween.TRANS_CIRC).set_ease(Tween.EASE_IN_OUT)
```

A estos se les puede indicar que función matemática utilizar y como aplicarla; la mayoría de animaciones son del tipo *TRANS_CIRC*, ya que después de probar varios tipos este era el que producía una animación más interesante. Estos *tweens* se configuran en modo paralelo para que ocurran todos en paralelo en vez de tener que esperar a que acabe uno para iniciar el siguiente.

Hay una instancia del menú en el que si que se han utilizado animaciones tradicionales, la primera pantalla que se ve al iniciar el juego. En esta se puede ver a un personaje corriendo por la pantalla y realizando una animación de bateo de vez en cuando. Ademas, cuando se quiere avanzar al menú principal, el personaje se zambulle dentro del menú con una animación.

Para realizar esto se ha tenido que añadir una cámara 3D en una perspectiva concreta junto con un personaje en 3D para que diera la sensación de estar dentro del menú. Las animaciones

de este personaje se han hecho con un *AnimationTree* completamente diferente al que se utiliza durante la partida.

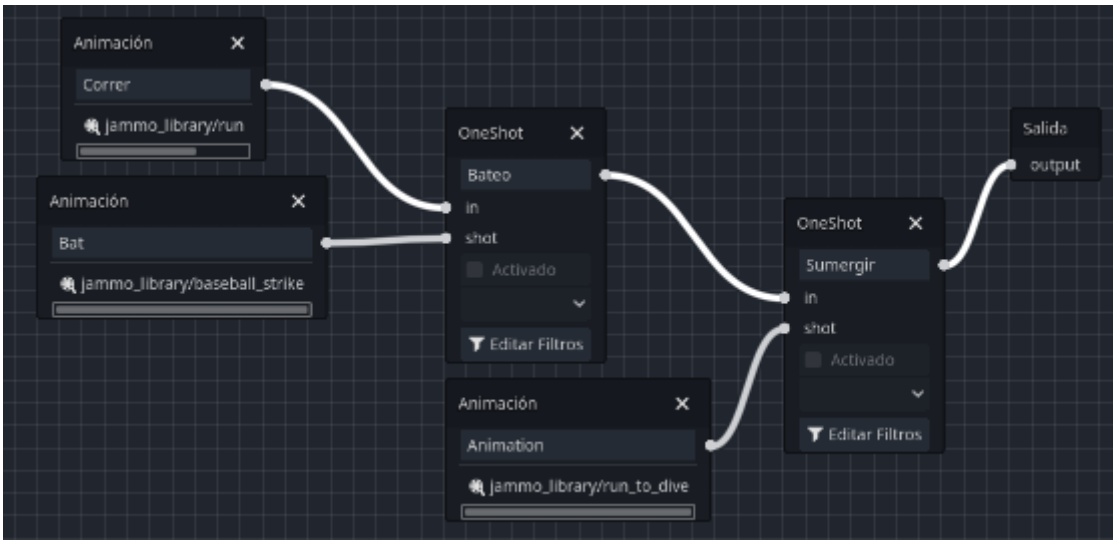

Figura 55: *AnimationNodeBlendTree* con las animaciones del personaje del menú

Aquí se ha utilizado un nodo del tipo *AnimationNodeBlendTree* para poder utilizar *OneShot*s. Estos se pueden ver en la imagen de la figura Figura 55 y tienen dos entradas: *in*, la animación que se reproducirá normalmente, y *shot*, una animación que se reproducirá una vez cuando se active el *OneShot* para después seguir reproduciendo la animación del *in.* Este *OneShot* se activa desde el *script* que maneja todas las animaciones del menú, es perfecto para el propósito que se busca aquí, tener al personaje caminando todo el rato y reproducir una animación de vez en cuando.

3.2.2.8 Cambio entre escenas

Tanto en este como en cualquier otro proyecto de Godot, las diferentes partes del juego están guardadas en escenas que se cargan y se descargan cuando es necesario. Tenemos una escena para el menú principal, una escena para el menú del multijugador, una escena para la partida de un jugador y una escena para una partida de multijugador, y el juego tiene que ser capaz de de cambiar entre estas escenas en todo momento.

La manera más sencilla de cambiar entre escenas es sustituir la actual por otra con:

```
get_tree().change_scene("res://path/to/scene.tscn")
```

Pero este acercamiento tiene un problema muy importante, y es que el juego puede quedarse bloqueado durante unos instantes durante el cambio de escena, lo que durante el desarrollo ha llevado a dos problemas: el primero es que este bloqueo hacia que no fuera posible mostrar una pantalla de carga mientras se esperaba a que cargase la partida, porque el juego se bloqueaba antes de enseñar esta pantalla y no se desbloqueaba hasta que la partida había cargado, dando por unos instantes la sensación de que el juego se te había bloqueado. La segunda es que como el menú esta dividido en dos escenas, al cambiar entre menús el pequeño bloqueo hacia que se notase el cambio entre escenas dando una sensación muy extraña al jugador.

Para solucionar esto hay que empezar a cargar en segundo plano la escena a la que queremos cambiar, esperar a que este completamente cargada y entonces decirle al juego que cambie a

la nueva. Esto aumenta el tiempo que se tarda en cargar, pero hace se pueda mostrar una pantalla de carga a la hora de empezar la partida y en el caso del menú hace que la transición sea completamente invisible para el usuario. Para realizar esta carga en segundo plano se puede utilizar la clase *ResourceLoader* de *Godot.* Con el siguiente comando se le indica que comience la carga.

```
ResourceLoader.load_threaded_get(SCENE_PATH)
```

Y después solo hay que comprobar cada fotograma si la carga ya se ha acabado y cargar la escena cuando sea correspondiente.

3.2.2.9 Ajustes

El usuario tiene que ser capaz de ajustar ciertas preferencias desde dentro del juego y que estas se guarden que cierre el juego y lo vuelva a abrir. Estas son: el nombre que utilizara en el multijugador, el volumen del sonido general del juego, de la música y de los efectos, el idioma de los textos del juego, la sensibilidad del giroscopio y si quiere activar o desactivar la vibración en el juego.

Para guardar todas estas preferencias en la memoria del móvil del usuario Godot incluye la clase *ConfigFile*, que crea un archivo de configuración que puede guardar variables en texto plano. Para poder manejar estas variables, se crea un script que se cargara en el a*utoload* y que al instanciarse lo primero que hace es comprobar si el archivo de configuración existe; en caso positivo carga el archivo, y en el negativo crea el archivo con unos valores predefinidos y después los carga.

Este archivo se llamara *config.cfg* y se guardara en *user://*, que es el espació de la memoria que el sistema operativo del teléfono le dedica a la aplicación. Cuando cargamos el archivo, guardamos todas las variables en un diccionario para que puedan ser fácilmente obtenidas desde cualquier otro script. Para facilitar el poder editar cualquier de estos valores, creamos una función que usaremos cada vez se encargara de cambiar el valor en el diccionario y asegurarse de volver a guardar el archivo de configuración con los valores actualizados.

Para aplicar estos ajustes tenemos un nodo con un *script* conectado a los botones que se encargan de cambiar estos ajustes, que lanzan una señal cada vez que cambian de valor. El script se dedica a aplicar el cambio de manera inmediata y de llamar a la función de guardar el valor en el *script* del párrafo anterior. También es el encargado de leer la configuración guardada en el diccionario de ese *script* y ponerle a los botones el valor adecuado.

3.2.3 Arte

3.2.3.1 Visuales

3.2.3.1.1 UI

Durante el desarrollo del juego se decidió diferenciar a los dos personajes de la partida haciendo que uno fuera de color rojo y otro de color azul, así que se decidió aprovechar el esquema de colores para el menú. Para tener algo más con lo que trabajar también se decidió utilizar el blanco para la letra y los fondos. El objetivo era que la estética fuese sencilla, basada en colores planos y contrastes.

Para la primera pantalla del juego combina la mitad superior de la pantalla en color blanco y la mitad inferior en color azul, con el nombre del juego en letras negras en la parte superior. En la parte superior, también en color azul, el personaje del juego.

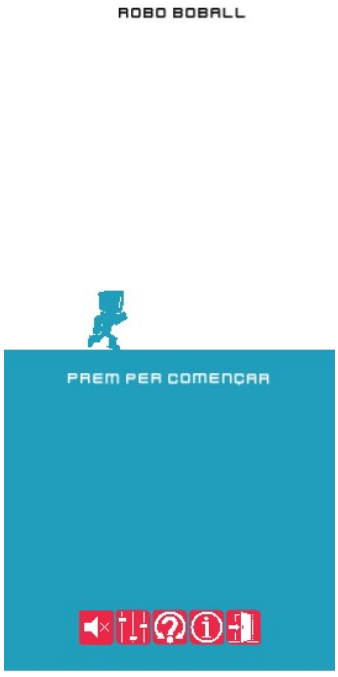

Figura 56: Pantalla inicial del juego

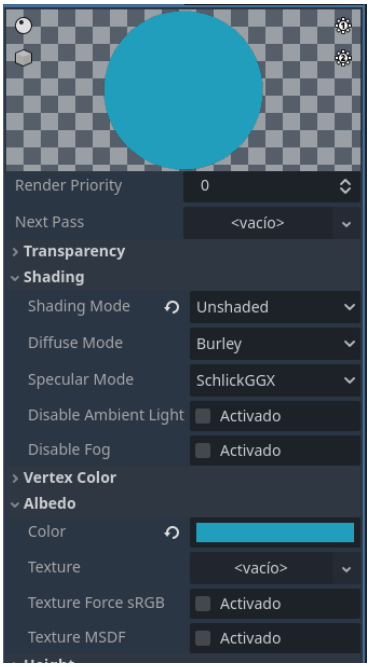

Figura 57: Material utilizado en el personaje de la pantalla inicial

El personaje este renderizado de tal manera que solo se ve su silueta y sus ojos. Para ello se usa un material que no realiza ningún tipo de sombreado en el modelo. Cuando se pulsa la pantalla para pasar al menú principal, el personaje se zambulle en la parte inferior de la pantalla y esta se desliza verticalmente hasta desaparecer por abajo.

En el menú principal hay dos franjas diagonales, una azul y una roja. Cuando se pulsa cualquiera de las dos esta se desliza verticalmente hasta terminar en la parte superior de la pantalla, sirviendo de titulo, mientras que la otra se desliza hacia fuera de la pantalla. Entonces, el menú correspondiente, que sera del color contrario al del botón que hayamos pulsado, entrara deslizando se ya sea de manera horizontal o vertical. Todo el menú sigue este mismo diseño de franjas diagonales que se deslizan a diferentes partes de la pantalla. El menú de configuración del juego también sigue este mismo patrón.

Figura 58: Menú principal del juego

Figura 60: Menú para entrar en una partida de un jugador

Figura 59: Menú para entrar en una partida multijugador

3.2.3.1.2 Personaje

El modelo 3D del personaje que se utiliza en el juego se llama Jammo y fue creado para uso libre como una colaboración entre los canales de *YouTube* «*Mix and Jam*» y «*Curiomatic*» *[31]* . El modelo incluye las texturas y viene completamente *riggeado*, preparado para poder utilizarse fácilmente con *Mixamo*.

El modelo 3D del bate que utiliza el personaje fue creado por el usuario de *OpenGameArt.org Lucian Pavel* e incluye texturas [32] .

Todas las animaciones utilizadas en el personaje han sido descargadas de *Mixamo* y editadas por mi para incluir en ellas el bate en las manos del personaje.

Para crear la variación de color del personaje en un inicio se utilizaba un *shader* que intercambiaba el canal *R* y *B* del la textura de *albedo* según una variable, pero al final acabo dando problemas por lo que se opto por crear una textura con esos canales intercambiados utilizando *Gimp* y tener dos materiales distintos, uno rojo y otro azul.

Para el material del personaje se buscaba que pareciera un juguete, que diera la sensación de ser de plástico y que brillara mucho con la luz del sol, así que lo primero fue activar el *rim,* que se podría traducir como borde, y aumenta el brillo en los puntos en los que el angulo de incidencia de la luz es menos elevado, provocando que las sombras interiores del modelo sean menos pronunciadas y que se vea más iluminado en general. El siguiente punto importante es el *clearcoat*, que se traduce como barniz y hace exactamente lo que su nombre indica, dándole al modelo un efecto de plástico brillante. Por ultimo, el material tiene una cantidad media de *metallic*, que regula como de metálico parece el material.

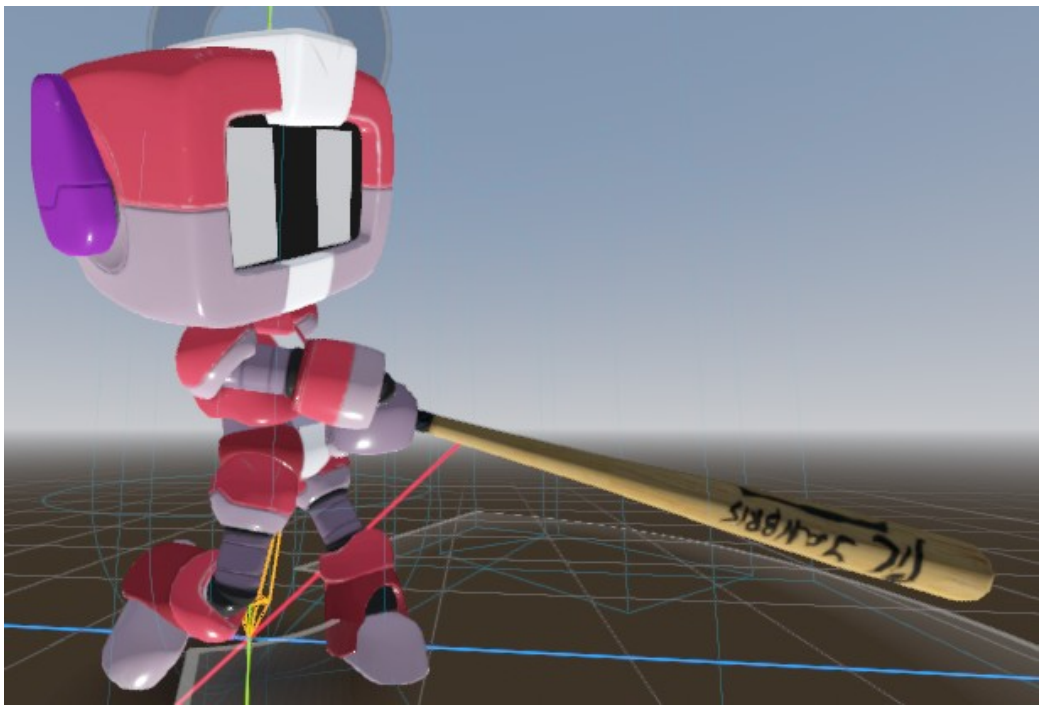

Figura 61: Modelo 3D de *Jammo* con el material aplicado en una animación de bateo

Como la sombra que producía Godot de manera dinámica no se veía especialmente bien, configure la malla del personaje para que no emitirá sombra y coloque un *sprite* debajo del personaje para imitar esta sombra. Este *sprite* es un circulo con un gradiente radial de negro a transparente para imitar una sombra suave.

Normalmente el bate es de tamaño pequeño, pero al inicio de la animación de bateo este se alarga hasta su tamaño natural, y al final de la animación este se vuelve a hacer pequeño.

Durante esta animación, el bate produce una estela. Esta se ha creado utilizando un *shader* de diseño propio programado con el editor visual de Godot. El primer paso para crear esta estela fue hacer su textura: un degradado vertical de azul a blanco, translucido y con los bordes completamente difuminados, que fue dibujado utilizando *Inkscape.* Para crear el efecto de estela se mueve rápidamente de manera horizontal la textura, y para ello se le aplica un *offset* en el eje *X* a sus *UV*. Este *offset* esta controlado por un parámetro del *shader*, que se utilizara en la animación de bateo para controlar la velocidad de la estela.

Para que el efecto funcione se tiene que forzar a la textura a no repetirse, así pasara de lado a lado rápidamente y desaparecerá. Para que el control del *offset* en la animación sea más sencillo, se mapean los valores para que estén entre uno y cero.

Figura 62: *Shader* de la estela visto en el editor visual de Godot

El personaje tiene encima de su cabeza un indicador circular que muestra el porcentaje de carga de su golpe. Para hacerlo se han utilizado dos texturas, una con forma de circulo con el centro de este transparente, haciendo forma de rosca, y otra igual pero que solo tiene coloreados los bordes mientras que el resto de la textura es transparente. Estas dos texturas están coloreadas en blanco, y gracias al nodo *ProgressTextureBar* de Godot podemos hacer que el indicador empiece como la textura vacía y se vaya rellenando de manera radial con la primera textura, pudiendo elegir de que color sera esta ultima.

Por ultimo, a los pies del personaje hay un indicador en forma de cono que indica hacia donde esta apuntando el personaje. Para crearlo, primero diseñe la textura en *Inkscape*, creando la forma con los bordes en color negro y rellena por un gradiente radial de negro a transparente. Para poder configurar más fácilmente este indicador no quería utilizar este indicador como la textura, si no como una mascara para poder hacer que el indicador tuviera la misma forma pero se pudiera cambiar de color desde el motor con un parámetro. Para ello se creo un *shader* con el editor de Godot que utiliza esta textura como el alfa pero utiliza el color que se seleccione como parámetro.

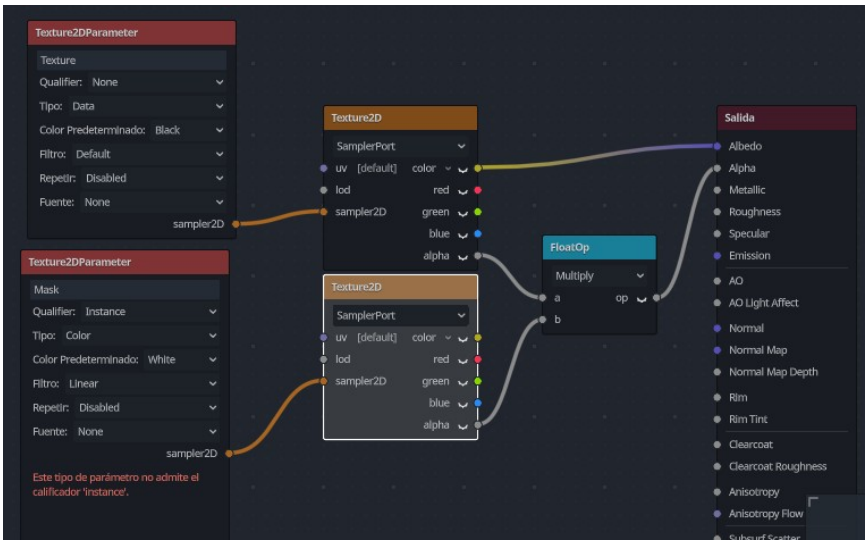

Figura 63: *Shader* del indicador visto en el editor visual de Godot

3.2.3.1.3 Campo

El campo en el que se juegan las partidas es simplemente un plano con una textura. Para poder diseñar correctamente las texturas cree el plano en *Blender*, hice el mapeo de las *UV* y lo exporte a Godot. La textura la diseñe en *Inkscape* intentando que fuera simple, con colores planos y algunos detalles en el área de cada jugador para aportar algo de textura. Se buscaba un diseño reminiscencia de una pista de futbol sala o de basquetbol pero más simple.

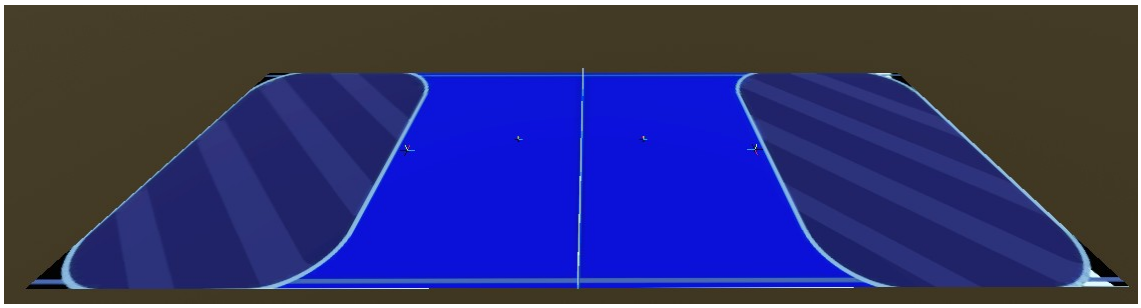

Figura 64: Pista del juego con solamente el plano a la vista

A cada lado de este pista se colocan un orbe flotante que actúa como portería, uno azul y otro rojo. Estos orbes son esferas con un *shader* aplicado, el cual no diseñe yo, sino el usuario de *godotshader.com «Miisannn»* [33] . Este hace que las esferas parezcan campos de fuerza, ayudando así a que el juego tenga un aire a ciencia ficción. El *shader* te permite modificar varios parámetros, como el color, como de distorsionado esta el campo o la frecuencia a la que vibra. Decidí exagerar bastante todos los valores para que el efecto se pudiese apreciar correctamente incluso en la pequeña pantalla de un móvil.

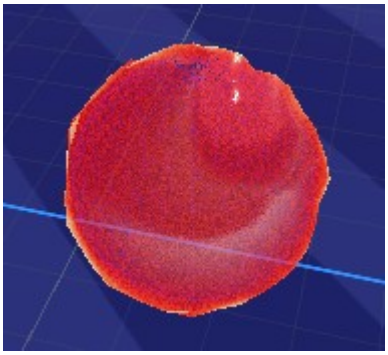

Figura 65: Imagen del orbe con el *shader* aplicado

Para añadir más dinamismo a la escena, ambos orbes tienen una animación en la que se bambolean ligeramente en vertical, para dar la sensación de que están flotando en el aire.

Para la pelota, diseñe en *Blender* un disco como el que se usa para jugar al *hockey* y le añadí en Godot un material con altos valores de *metallic* y un tinte rojo claro. Este disco también tiene una animación de bamboleo como las porterías y su sombra es un *sprite* al igual que como la del personaje.

Para decorar los exteriores del campo, la parte en la que no se juega, decidí utilizar un paquete de modelos 3D gratuitos creados por *Kenny,* en concreto el Kit Estación Espacial [34] para crear una estética futurista. Para colocar estos modelos utilice el sistema de *GridMap* de Godot, que te permite colocarlos utilizando una cuadricula con varios niveles de altura y hace que sea muy fácil montar un escenario sencillo siempre que todas las piezas sean del mismo

tamaño. Los coloque rodeando el campo, añadiendo decoración en la parte inferior y superior, que iban a ser las partes que se pueden ver correctamente si el juego se ejecuta en un móvil.

Por ultimo quedaba añadir luces y configurar el entorno para crear una iluminación que favoreciese el conjunto. Como se busca una estética estilizada y poco realista se configuro el entorno para utilizar una luz ambiente potente, con ajustes para aumentar la saturación y una corrección de color para hacer los colores más vividos. Las luces se configuraron para utilizar principalmente luz directa en vez de indirecta.

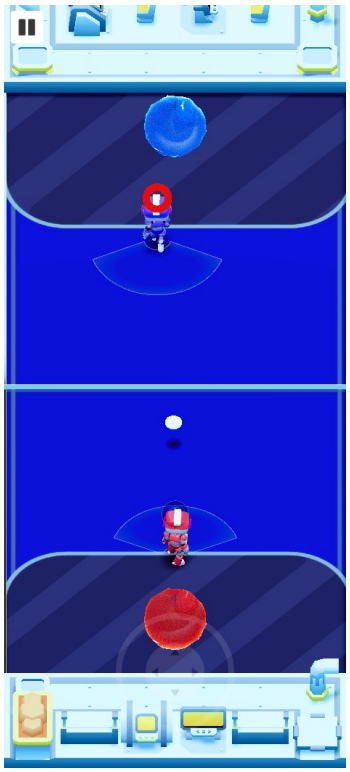

Figura 66: Juego corriendo en una partida con el campo completamente acabado.

3.2.3.2 Sonido

3.2.3.2.1 Música

En el juego se utilizan dos pistas de música, una para el menú y otro para la partida, independientemente de si es de un jugador o multijugador.

Las dos canciones son del mismo autor, y se venden el mismo paquete, llamado «*Retro and Electronic Music Pack» [35]* . Al comprar este paquete obtienes permiso para utilizar las canciones para cualquier uso ya sea personal o comercial. Elegí este paquete porque las canciones suenan muy a videojuego y tienen una estetica futurista o de ciencia ficción.

Para el menú elegí la canción *Groove Dungeon* ya que es pegadiza y tiene ritmo pero a la vez es lo suficientemente calmada para poder escucharla en bucle en el menú y que no se haga pesada, además de que es relativamente plana.

Para la partida use *ElectroFest*, una canción con más movimiento, que va subiendo mientras avanza la canción pero que vuelve a bajar al llegar al final para que se pueda repetir sin problemas.

3.2.3.2.2 Efectos de sonido

Los efectos de sonido utilizados vienen de varias fuentes. Primero, el efecto que suena cuando se pulsa un botón en el menú viene del paquete *UI Audio* de *Kenney* [36] , que contiene muchos tipos de sonidos para interfaces de usuario. Seleccione un sonido de «*click*» corto y que no destacara demasiado para que no se hiciera pesado escucharlo cada vez, pero lo suficiente llamativo para que se notase y ayudara a mejorar la sensación de pulsar el botón.

Para el sonido que suena cuando se batea y cuando se golpea la pelota utilice un video de *GFX Sounds,* un canal de *YouTube* que se dedica a subir efectos de sonido [37] . El video tenia varios sonidos de bateo y de una pelota siendo golpeada, así que decidí usar los diferentes sonidos de bateo para los diferentes niveles de potencia del golpe del jugador. Para separarlos me descargue el video lo edite con *Audacity* para poder exportar cada uno de los efectos por separado.

El sonido que suena cuando la pelota entra en una de las porterías lo encontré en *Soniss*, una web que vende efectos de sonido y que tiene una gigantesca librería de efectos de sonido para videojuegos gratuita, que cada año coincidiendo con la *GDC* añade nuevos sonidos [38] . Seleccione este sonido porque sonaba futurista y suena como una pequeña explosión, lo cual tiene sentido teniendo en cuenta que las porterías son campos de fuerza.

3.2.4 Localización

Todos los textos del juego han sido localizados a Catalán, Castellano e Ingles.

Godot tiene un sistema de localizaciones automatizado que hace que añadirlas a un juego sea una tarea muy sencilla. Simplemente hay crear un archivo *csv*, que se puede crear con cualquier programa de hojas de calculo, yo utilice *LibreOffice*.

Hay que crear una columna para las *keys*, que sera una palabra que actuara como equivalente de la frase traducida en el motor. Después, hay hacer una columna para cada idioma al que se quiera traducir, y poner en cada fila el la frase traducida.

key	en	es
TAP_START	[center][shake rate=20.0 level=6 connected=1]Tap to start[/shake][/center]	[center][shake rate=20.0 level=6 connected=1]Pulsa para empezar[/shake][/center]
TITLE_SINGLEPLAYER	[center]Singleplayer	[center]Un jugador
TITLE_MULTIPLAYER	[center]Multiplayer	[center]Multijugador
TEXT_SINGLEPLAYER	[center]Play agains a machine in 1vs1 matches	[center]Juega contra la maquina en partidos de 1 contra 1

Figura 67: Archivo *csv* con las diferentes traducciones del juego

En el motor, en vez de escribir el texto quieres que se vea, escribes lo que hayas puesto en la columna *key*, y cuando inicies el juego ese texto se cambiara por el texto traducido. Por defecto Godot traduce automáticamente todos los nodos en los que se pueda escribir texto siempre que se encuentre una traducción disponible, pero cada nodo tiene una opción para activar o desactivar la traducción. Además, Godot te muestra el juego en el idioma correspondiente al de tu sistema operativo (siempre que la traducción exista), pero se puede cambiar en cualquier momento el idioma a mostrar vía código.

En este proyecto no ha hecho falta utilizarlo, pero si hay que traducir algo que no sea texto, como una imagen o un archivo de audio, Godot trae un sistema para configurar esas sustituciones y que cambien automáticamente según el idioma como el resto de la traducción.

4 Pruebas

Entre todas las sesiones de pruebas realizadas, sin contarme a mi mismo, han participado diez personas de diferentes edades y con diferentes niveles de experiencia jugando a videojuegos.

4.1 Prototipos

Los dos prototipos de este proyecto nunca han llegado a entrar en una etapa de pruebas formal, así que las diferentes pruebas las he realizado yo mientras lo desarrollaba. El principal objetivo de estas pruebas era ver si las mecánicas que estaba implementando estaban funcionado correctamente. Además, también valoraba como de divertidas eran y como de complejas o difíciles de entender podían llegar a ser.

Otras personas vieron y probaran estos prototipos, pero solo de manera muy general, dando ideas de como podrían evolucionar esas mecánicas o que partes creían que funcionaban mejor.

En general, durante esta fase no me preocupe especialmente por el *feedback* externo, ya que sabia que la mayor parte de lo que se podía jugar no iba a ser final e iba a cambiar mucho en las siguientes etapas del desarrollo, y tenia confianza en ser capaz de identificar yo mismo que mecánicas tenían sentido para el futuro del desarrollo.

4.2 Fase inicial del desarrollo

Esta se sitúa con el prototipo completamente terminado y con unas pocas semanas de desarrollo. Ya era posible jugar una partida en multijugador de inicio a fin. En total, seis personas probaron el juego en este estado, jugando partidas entre ellas. Estas pruebas sirvieron principalmente para encontrar una gran cantidad de errores en diferentes partes del juego, desde la lógica del multijugador a cosas que se me habían pasado por alto en las mecánicas básicas.

Al ser un juego pensado para dos personas (en este momento el modo de un jugador no estaba implementado) es muy difícil probarlo correctamente y fijarse en los errores siendo solo una persona, así que estas sesiones fueron increíblemente útiles para detectar errores que de otra manera me habrían sido imposibles de encontrar.

Como en estos momentos el juego aun se encontraba en un estado muy poco avanzado, estas pruebas se centraron más en encontrar errores importantes que en recabar ideas y *feedback* sobre las mecánicas o sobre como mejorar el juego.

4.3 Fase media del desarrollo

En estos momentos el modo de un jugador ya ha sido implementado, aunque la inteligencia artificial del rival aun es muy básica como para poder tener una buena experiencia jugando este modo, así que las pruebas se centraron en el multijugador.

En estas pruebas participaron ocho personas, cuatro de ellas que ya habían participado en las pruebas anteriores. En estos momentos los errores más importantes ya habían sido corregidos, así que las partidas podían transcurrir con cierta normalidad. Seguían apareciendo errores, algunos que ya había encontrado previamente mientras desarrollaba el juego pero que no había sido capaz de reproducir, y que gracias a estas sesiones fui capaz de solucionar más adelante.

4.3.1 Problemas

Durante estas pruebas me asegure de fijarme en cual eran los puntos de fricción con los jugadores, que partes no se entiendan correctamente, que mecánicas eran más difíciles de dominar, y en general preguntar la opinión de la gente que había jugado y que cosas creen que se podrían cambiar. En esta lista se ven los puntos que más se repitieron entre todos los jugadores:

- El apuntado no es cómodo de utilizar. Cuesta mucho decidir conscientemente hacia donde se enviara la bola. Mucha gente no acaba de entender como funciona hasta que se lo explico.
- Es demasiado fácil hacer un bateo sin querer. Sobre todo cuando el *stick* virtual toca su zona muerta.
- El tiempo que pasa entre que te dejas de mover y empiece a batear es demasiado alto.
- Es muy difícil darle a la pelota.
- El tiempo que tardas en cargar un golpe es demasiado alto, te deja muy poco tiempo para reaccionar.
- El personaje se mueve demasiado lento.
- Las animaciones no reflejan correctamente el movimiento del personaje.

Todos estos problemas me hicieron replantear como funcionaban las mecánicas, ya que para muchas de estas tenia más sentido replantarlas que retocarlas.

4.3.2 Soluciones

La primera queja, la del apuntado, me hizo pensar en cambiar enteramente como se apuntaba, ya que en este momento la dirección a la que se golpeaba la pelota era la misma a la que el personaje se estuviera moviendo justo antes de pararse. Fue entonces cuando se me ocurrió separar completamente el movimiento del apuntado, y eso fue lo que hizo que se me ocurriera utilizar el giroscopio para apuntar.

Para el problema de que se bateaba sin querer, implemente que al detectarse que el personaje se deja de mover, se esperaran unos pocos fotogramas de margen antes de batear. Además, hice que en cuanto este margen de tiempo acabara se empezara el bateo, con una animación más rápida que antes para facilitar el poder darle a tiempo a la pelota.

Las quejas sobre el tiempo de carga también me hicieron replantearme como funcionaba este sistema, que acabe cambiando para que en vez de tener que llenar una barra de carga que lleve mucho tiempo se llenaran tres, la primera llenándose muy rápidamente pero haciendo que la pelota se golpee con poca fuerza, y las siguientes tardando más tiempo pero también golpeando más fuerte.

El problema de las animaciones no le hice mucho caso porque los personajes, y por tanto las animaciones que estaba utilizando en ese momento eran temporales y iban a ser sustituidas eventualmente.

Por ultimo, aumente la velocidad del movimiento base del personaje e hice que al moverse hacia la dirección de la portería la velocidad aumentase, para que fuera más fácil parar las pelotas que dirigidas a esta.

4.4 Fase final del desarrollo

En esta fase todas las mecánicas estaban implementadas, gran parte de los gráficos del juego eran finales, tanto el modo multijugador como el de un jugador eran jugables (aunque con algún que otro error) y se buscaba solucionar problemas más sutiles. Faltaba toda la parte gráfica de los menús y todos los sonidos.

En estas pruebas también participaron ocho personas, algunas que ya habían participado en la primero prueba, y otras de la segunda prueba. La gente probo los dos modos de juego, y al igual que en las pruebas anterior, fui observando sus partidas y preguntando sobre posibles mejoras o cosas que no les hubieran gustado.

4.4.1 Problemas

La siguiente lista contiene los principales problemas que salieron a la luz en esta sesión:

- El sistema de apuntado con el giroscopio no gusta a todos los jugadores
- Es demasiado fácil apuntar en linea recta por el centro del campo y marcar gol.
- La pelota va demasiado rápido.
- Las partidas son muy cortas
- El rival controlado por IA es demasiado bueno jugando.
- Algunos jugadores no acabaron de entender el sistema de cargas.
- El personaje tarda mucho en volver a poder moverse después de batear.
- El cambio entre la animación de moverse y de batear es demasiado abrupto
- El indicador de dirección de bateo no es muy preciso.

4.4.2 Soluciones

Para el primer problema, cuando decidí implementar el giroscopio ya sabia que habría gente a la que no le gustaría y a la que les costaría acostumbrarse, ya que no es un metido de control

muy común y ademas al mover el móvil estas moviendo la pantalla que estas mirando, lo que puede resultar bastante confuso. Para mitigar esto, decidí hacer que se pudiera configurar la sensibilidad del movimiento con el giroscopio, para que cada jugador pueda decidir cuanto necesita girar el móvil, adaptándose así a la gente que prefiere no tener que mover demasiado el móvil.

Para el segundo, implemente un sistema por el cual mientras más alejada del centro del indicador de bateo estuviera la pelota, más desviada iría, haciendo que aun fuera posible enviar las bolas rectas pero necesitando ser mucho más preciso para que acabaran llegando a la portería.

El problema con la velocidad de la pelota no lo había notado ya que ya estaba acostumbrado a jugar a esa velocidad, pero claramente era demasiado rápida para que la mayoría de jugadores pudieran reaccionar a tiempo. Para arreglar esto, cree un menú de configuración de partida en el que poder elegir como de rápido fuese la bola, permitiendo así tres niveles de de dificultad según la velocidad de la bola para acomodar a distintos tipos de jugadores. En este mismo menú añadí también un selector para poder determinar cuantos puntos se tienen que obtener para ganar la partida, pudiendo así tener partidas más cortas o más largas.

El jugador controlado por IA no siempre golpea la bola cuando se la lanzas, pero cuando lo hacia siempre apuntaba a la portería. Cambie este comportamiento para priorizar los lanzamientos rebotados para darle al jugador más margen para devolver la bola.

El sistema de cargas tiene varios colores que indican la fuerza del golpe, pero este sistema de colores que a mi me resultaba muy intuitivo parecía no ser universal, ya que no todo el mundo lo interpretaba de la misma manera. Por ello, al hacer un bateo, según el nivel de carga la tanto la cámara como el propio móvil vibrasen con más intensidad según lo alto que fuese el nivel de carga. Ademas también decidí que cada nivel tendría un sonido de bateo diferente, para ayudar a diferenciarlos más.

Los dos siguientes problemas los solucione juntos, ya que el personaje no se mueve hasta que la animación de golpeo ha terminado completamente y se ha vuelto a la animación de correr, hice que la transición entre estas dos fueran más rápida y configure el *blending* entre las dos para que el paso entre animaciones fuese más natural.

El ultimo error nunca se llego a solucionar como tal, ya que aunque hice varias pruebas para mejorar el indicador y hacerlo más preciso, añadiendo partes extras que indicaran la trayectoria de la pelota, decidí dejarlo como estaba porque hacia que el intentar apuntar la pelota hacia donde se quería fuese más divertido.

5 Conclusiones

5.1 Conclusiones personales

Después de terminar este proyecto puedo afirmar que el resultado ha sido completamente satisfactorio.

Mi objetivo personal con este proyecto era desarrollar un videojuego de principio a fin y que se sintiera **terminado**, completamente jugable y sin concesiones, ya que en mi anterior proyecto reaccionado con los videojuegos (el trabajo final de la asignatura de Diseño de Videojuegos) sentí que había intentado abarcar mucho más de lo que podía hacer, y que debido a ello todos los aspectos del juego quedaron a medias.

También tenia gran interés en aprender a utilizar Godot, ya que en el proyecto anteriormente mencionado acabe muy desencantado con Unity, y la propuesto de funcionamiento sencillo y código abierto que pone sobre la mesa Godot me atraía mucho.

Durante el desarrollo de este proyecto he aprendido mucho de diseño de videojuegos, de organización (tanto a nivel de código como a nivel personal), de como funciona Godot y de como proceder en un desarrollo en solitario de un videojuego.

Me parece que el acercamiento que he tomado en cuanto al prototipado me ha ayudado enormemente a acabar consiguiendo un resultado satisfactorio, ya que me ha permitido descartar las ideas y conceptos que no me funcionaban antes de que se supusieran una inversión de tiempo demasiado grande y ademas ha servido para comprobar que cosas no hay que hacer al intentar implementar ciertos elementos. Esto ha sido de gran importancia, porque al empezar con el desarrollo del proyecto final he tenido unas bases muy solidas ya que he evitado repetir los problemas que me había encontrado durante el prototipado.

Haber sido capaz de implementar un modo multijugador en linea me parece una de las grandes victorias de este proyecto, ya que cuando me plantee la idea de hacerlo no estaba seguro de que fuera a ser capaz. En su estado actual, el multijugador funciona correctamente y es muy divertido, aunque tiene ciertos problemas si se quiere jugar a través de internet. Todo el proceso ha sido muy didáctico y me ha hecho ganar experiencia al respecto que estoy seguro de que me sera muy útil en futuros proyectos.

Todo el proceso del desarrollo de este proyecto me ha ayudado a ganar soltura en muchos ámbitos, especialmente en el diseño. Afrontar yo solo desde el planteamiento hasta la implementación del juego significa que tengo total control creativo sobre todos y cada uno de los aspectos del juego, pero eso también significa que tengo responsabilizarme de todos ellos. Esto ha hecho que tenga que aprender por las malas que partes del diseño funcionan y que partes no lo hacen, además de enseñarme una lección muy valiosa: a veces hay que dar marcha atrás y replantearse lo que uno esta haciendo, aunque eso significa perder muchas horas de trabajo.

El primer prototipo que se hizo para este proyecto tiene varias mecánicas que fueron muy difíciles de implementar y en las que invertí mucho esfuerzo y tiempo, pero no me arrepiento ni un poco de tirar ese proyecto y empezar de cero, ya que todas las lecciones que aprendí mientras lo hacia hicieron que el juego final fuera aun mejor.

Siguiendo con el diseño, estoy muy satisfecho con como he podido coger unas mecánicas básicas y sencillas y darles una vuelta de tuerca para crear algo, igualmente básico, pero más único y con cierta personalidad. Creo que trabajar con limitaciones autoimpuestas en el esquema de control me ha hecho ser más creativo y pensar más en que quería implementar y en como quería hacerlo.

No podría acabar este apartado sin hablar de la principal herramienta utilizada en este proyecto, Godot, que me ha sorprendido gratamente. En un inicio, sobre todo viniendo de Unity, muchas de las funciones del motor pueden parecer contra-intuitivas, pero cuando te acostumbras a su filosofía de diseño y aprendes como espera el motor que interactúes con el descubres lo versátil que es, todas las herramientas que tiene para facilitar tu tarea como desarrollador y como ha sido desatollado priorizando en todo momento la simplicidad.

Durante todo el desarrollo he utilizado la rama *beta* de Godot, que ha recibido actualizaciones periódicas cada mes, muchas de ellas incluyendo o mejorando funciones que me han facilitado diferentes partes del desarrollo. Estar en esta rama también ha hecho que encuentre algunos errores graves en el programa, los cuales he reportado en el repositorio oficial y han sido solucionados a los pocos días [39] [40] .

Como conclusión, estoy muy satisfecho con todo el trabajo realizado en este proyecto, creo que ha resultado muy didáctico y que todo lo que he aprendido aquí me sera útil en un futuro y tengo muchas ganas de utilizar todos los conocimientos adquiridos aquí en futuros proyectos.

5.2 Lineas de futuro

Si continuara con el desarrollo de este proyecto hay varios aspectos que me gustaría ampliar y mejorar.

Primeramente, aunque para este proyecto estoy contento con cierto minimalismo en cuanto a las mecánicas del juego, si lo continuara me gustaría añadir algunas nuevas. Metería más modos de juegos para variar la experiencia, como por ejemplo algún modo en el que el escenario cambie y tenga obstáculos, o que haya diferentes *power-ups* esparcidos para dar ventaja a los jugadores que se arriesgasen a ir a por ellos. También estaría bien añadir más personajes jugables, que tuvieran algunas ligeras diferencias visuales y mecánicas entre si para conseguir que la experiencia de juego se mantuviese fresca por más tiempo.

En cuanto al multijugador, seria imprescindible centrarse en adecuarlo más al multijugador vía internet, ya que ahora mismo es muy fácil que el cortafuegos del dispositivo deniegue los paquetes del juego si no esta configurado por el usuario. Ademas, añadiría un buscador de partidas a nivel global, para no tener que introducir la dirección de *IP* de servidor cada vez que quieres unirte a una partida en un wifi diferente al tuyo. Todo esto se podría solucionar fácilmente teniendo un servidor central que maneje todo el trafico de red del juego.

Un punto que en un inicio iba a formar parte de este trabajo pero que se descarto a medio camino fue la publicación del juego, ya que la idea inicial era publicarlo en las plataformas oficiales de *Android* y *IOS.* Se acabo descartando por falta de tiempo y de recursos, ya que la publicación tiene una serie de costes tanto de tiempo como monetarios, sobre todo si te la

tomas en serio. Hablando de *IOS,* desarrollar un *port* del juego a esa plataforma también seria un punto muy importante, ya que en un juego multijugador tiene que poder jugar la mayor cantidad de gente posible.

Desarrollar una versión para plataformas de sobremesa queda completamente descartado, ya que todas las decisiones de diseño del proyecto están están profundamente enraizadas en el diseño móvil.

El ultimo cambio que se le haría al juego seria sustituir todo el arte de terceros por arte propio, ya fuese hecho por mi o por una persona contratada para hacerlo, ya que eso ayudaría al juego a tener una estética y personalidad propia .

5.3 Costes

Todas las herramientas utilizadas en este proyecto (Godot, Blender, Inkscape, Gimp, Neovim) son gratuitas y de código abierto, así que no han tenido ningún coste asociado.

Todos los recursos visuales utilizados han sido de obtenidos de paquetes de recursos gratuitos o han sido subidos a paginas para ser utilizados de manera gratuita, así que tampoco han tenido ningún coste.

El paquete de música que he utilizado [35] se vende actualmente por $19,99USD, aunque yo lo compre hace dos años en un *bundle* de *itch.io [41]* que incluía este y otros cientos de juegos y recursos para videojuegos, así que su coste se podría considerar negligible.

En cuanto al coste temporal, desde que descargue Godot en julio de 2023 hasta que entrego este proyecto en septiembre de 2024 le he dedicado un total de 640 horas al trabajo.

6 Entrega

El binario de instalación para Android se encuentra en la pagina oficial del juego en *itch.io: https://colladog.itch.io/robo-bobot/*

Todo el código fuente del proyecto se encuentra en el siguiente repositorio de *github: https://github.com/a-collado/tfg-juego-2/*

7 Referencias

[1] Viveros, M.C., García, D. (2009). Elaboración de una guía para el desarrollo de aplicaciones en extjs. Instituto Tecnológico de Orizaba

[2] UnirFP. (2022, Septiembre) Framework: qué es, para qué sirve y algunos ejemplos [Online] https://unirfp.unir.net/revista/ingenieria-y-tecnologia/framework/

[3] Wikipedia. (2024) AAA (industria del videojuego) [Online] https://es.wikipedia.org/wiki/AAA_(industria_del_videojuego)

[4] Atari 2600 Specifications [Online] https://problemkaputt.de/2k6specs.htm

[5] Williams, Andrew (2017, Marzo). History of Digital Games: Developments in Art, Design and Interaction. CRC Press.

[6] GameSpot (2014, Marzo). Classic Studio Postmortem: Lucasfilm Games [Online] https://www.youtube.com/watch?v=HDvEFbh6I2g

[7] Fiadotau, Mikhail (2019). "Dezaemon, RPG Maker, NScripter: Exploring and classifying game 'produsage' in 1990s Japan". Journal of Gaming & Virtual Worlds.

[8] Marcus Toftedahl (2019, Septiembre). Which are the most commonly used Game Engines? [Online] https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-

[9] Unreal Engine (2023). Licensing [Online] https://www.unrealengine.com/en-US/license

[10] Unity Manual (2023*). Render pipelines [Online] https://docs.unity3d.com/Manual/render-pipelines.html*

[11] Bonfiglio, Nahila (2018, Octubre). "DeepMind partners with gaming company for AI research". The Daily Dot.

[12] Unity (2023). Pricing updates. [Online] https://unity.com/products/pricing-updates

[13] Wikipedia (2024, Septiembre). Sprite (videojuegos). [Online] https://es.wikipedia.org/wiki/Sprite_(videojuegos)

[14] Christian, Brian; Isaacs, Steven (2015, Diciembre). GameMaker Programming By Example. Packt Publishing Ltd.

[15] GameMaker (2024). Licencing. [Online] https://gamemaker.io/en/get

[16] Godot (2024). List about features. [Online] https://docs.godotengine.org/en/stable/about/list_of_features.html

[17] Godot-Languages-Support (2023). Godot Languages Support. [Online] https://github.com/Godot-Languages-Support/godot-lang-support

[18] Valve Developer Community (2024). Shader. [Online] https://developer.valvesoftware.com/wiki/Es/Shader

[19] mxgmn (2024). WaveFunctionCollapse. [Online] https://github.com/mxgmn/WaveFunctionCollapse

[20] Wikipedia (2024, Septiembre). Incremental game. [Online] https://en.wikipedia.org/wiki/Incremental_game

[21] Battery Acid Dev. Youtube. [Online] https://www.youtube.com/@BatteryAcidDev

[22] Godot Docs (2024). High-level multiplayer. [Online] https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html
[23] Godot Docs (2024). SubViewport. [Online] https://docs.godotengine.org/en/stable/classes/class_subviewport.html
[24] Wikipedia (2024, Agosto). Cambio de base. [Online] https://es.wikipedia.org/wiki/Cambio_de_base
[25] Wikipedia (2024, Agosto). Búsqueda de ruta. [Online] https://es.wikipedia.org/wiki/B%C3%BAsqueda_de_ruta
[26] Volodymyr Tsaruk (2023, Septiembre). How do multiplayer games work? From simple to complex. [Online] https://gamestudio.n-ix.com/how-do-multiplayer-games-work-from-simple-to-complex/
[27] Frankie MB (2021, Septiembre). Rollback Netcode, el revolucionario sistema predictivo con el que jugar online retoma la gloriosa sensación de compartir recreativa. [Online] https://www.vidaextra.com/accion/rollback-netcode-revolucionario-sistema-predictivo-que-jugar-online-como-compartir-recreativa
[28] Godot Docs (2024). High level multiplayer. [Online] https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html
[29] Wikipedia (2024, Julio). Vector (informática). [Online] https://es.wikipedia.org/wiki/Vector_(inform%C3%A1tica)
[30] Godot Docs (2024). ENetConnection. [Online] https://docs.godotengine.org/en/4.3/classes/class_enetconnection.html
[31] Mix and Jam (2019). Jammo-Character. [Online] https://github.com/mixandjam/Jammo-Character
[32] Lucian Pavel (2016, Noviembre). Wooden Bat. [Online] https://opengameart.org/content/wooden-bat
[33] Miisannn (2024, Julio). 3D Bubble/Spatial shield shader. [Online] https://godotshaders.com/shader/3d-bubble-spatial-shield-shader-2/
[34] Kenney (2024, Mayo). Space Station Kit. [Online] https://www.kenney.nl/assets/space-station-kit
[35] Slaleky . Retro and Electronic Music Pack. [Online] https://slaleky.itch.io/retro-and-electronic-music-pack
[36] Kenney (2012, Octubre). UI Audio. [Online] https://www.kenney.nl/assets/ui-audio
[37] GFX Sounds (2023, Mayo). Baseball Bat Sound Effect. [Online] https://www.youtube.com/watch?v=JHjvabhUDVU
[38] Sonniss. Royalty Free Sound Effects Archive: GameAudioGDC. [Online] https://sonniss.com/gameaudiogdc
[39] a-collado (2024, Julio). Godot Editor freezing after playing scene only on Wayland on Godot 4.3 Beta 3. [Online] https://github.com/godotengine/godot/issues/94228

[40] a-collado (2024, Julio). Rendering errors on Mobile and Compatility renderer using Godot 4.3 Beta 3. [Online] https://github.com/godotengine/godot/issues/94225

[41] Necrosoft Games (2022). Bundle for Ukraine. [Online] https://itch.io/b/1316/bundle-for-ukraine

8 Ilustraciones

Figura 66: Juego corriendo en una partida con el campo completamente acabado.

Figura 67: Archivo csv con las diferentes traducciones del juego