Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

BRAWLCHEMY

Alumno/Alumna: López Santamaría, Guzmán

Director/Directora (1): Villamañe Gironés, Mikel

Curso: 2018-2019

Fecha: Bilbao, 24, junio, 2019

Resumen

En una sociedad en la que la tecnología es cada vez más común, crece continuamente el uso de ésta como método de entretenimiento por gran parte de la población. Y en el universo del entretenimiento encontramos el mundo de los videojuegos. Sin embargo, para muchas personas interesadas en entrar por primera vez a este mundo, éste puede ser extremadamente abrumador, pues la cantidad de opciones entre las que elegir es prácticamente ilimitada.

Brawlchemy pretende ser un punto de entrada en el mundo de los videojuegos, particularmente en el subgénero de los MOBA. Si bien éste es uno de los subgéneros más populares en la actualidad, también es uno de los más complicados de entender para nuevos jugadores. Por ello, Brawlchemy tratará de ser la versión más simplificada posible del género, para así poder disfrutarlo de la forma más básica sin necesidad de ser un experto.

Abstract

In a world where technology is more and more common, it is increasingly used as an entertainment by a large part of the population. And in the universe of entertainment we can find the world of videogames. However, for many people interested in entering this world for the first time, it can be extremely overwhelming, since the number of options to choose from is almost unlimited.

Brawlchemy aims to be an entry point into the world of videogames, particularly in the MOBA subgenre. While it is one of the most popular subgenres at the moment, it is also one of the most difficult to begin with. Therefore, Brawlchemy will try to be the most simplified version of the genre, to be able to enjoy the basics without the need to be an expert in the field.

Laburpena

Teknologia gero eta gehiago ohikoagoa den mundu batean, gero eta populazioaren entretenimendu gisa erabiltzen da. Eta entretenimendu unibertsoan, bideojokoen mundua aurki dezakegu. Hala ere, mundu honetan lehen aldiz sartu nahi duten jende askorentzat oso erabatekoa izan daiteke, aukeratzeko aukeren kopurua ia mugarik gabe dagoelako.

Brawlchemy-k bideojokoen mundura sartzeko puntua izan nahi du, bereziki MOBA azpigenerrean. Une honetan azpigororik ezagunenetakoa da, baina hasierako zailenetako bat ere bada. Beraz, Brawlchemy generoaren bertsio sinplifikatuena izango da, oinarriak gozatzeko gai izan dadin arloan aditua izan beharrik izan gabe.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

Hace no tantos años los ordenadores eran grandes, caros y utilizados solamente por profesionales. Hoy, sin embargo, hemos llegado al punto en que prácticamente todos llevamos en el bolsillo un «ordenador» mucho más potente de lo que nunca hubiésemos imaginado. La tecnología ha pasado rápidamente al plano del entretenimiento, y es en este plano donde los videojuegos han encontrado la popularidad que tienen actualmente.

Dentro del mundo de los videojuegos existen multitud de subgéneros. Para aquellos para los que este mundo sea un hobby, puede parecer sencillo entender las cualidades y peculiaridades cada uno de estos géneros y las diferencias entre ellos. Para quienes estén adentrándose en él, tanta cantidad de información puede ser extremadamente abrumador, por lo que muchos abandonan antes de poder encontrar un género que les guste. Y, si bien algunos géneros son bastante accesibles de cara a nuevos jugadores, existen muchos otros que requieren un alto nivel de conocimientos y habilidades previas. Es aquí donde entra en juego este proyecto, Brawlchemy.

Brawlchemy pretende ser un punto de entrada en el subgénero de los MOBA. Si bien éste es uno de los subgéneros más populares en la actualidad, también es uno de los más complicados de entender para nuevos jugadores. Brawlchemy tratará de ser la versión más simplificada posible del género, eliminando todos los elementos secundarios y manteniendo todos aquellos que lo hacen único y extremadamente entretenido, para así poder servir como una prueba de todo lo que este subgénero ofrece sin requerir un esfuerzo al jugador.

1.1. Razones para la elección del TFG

Uno de los principales motivos por los que se ha elegido realizar este trabajo es por la gran variedad de tecnologías y conocimientos previos que son necesarios. Para el desarrollo de cualquier videojuego con cierta profundidad es necesario aplicar la gran mayoría de las competencias que se supone han sido adquiridas a lo largo de los estudios del grado: desde los aspectos básicos y avanzados de la programación, pasando por las matemáticas que se encarguen de las físicas y hasta la gestión de bases de datos que puedan ser utilizadas, entre muchas otras. Todo esto hace que el proyecto sea claramente atractivo tanto para probarme a mí mismo como para demostrar a los demás el amplio abanico de competencias de las que dispongo.

Por otro lado, un buen TFG puede servir como carta de presentación para un ámbito tanto laboral y/o profesional como también educativo de cara a unos estudios superiores tras terminar el grado. En el caso en que se tenga claro el camino por el que uno quiere enfocar su futuro, es apropiado que el TFG se aproxime lo más posible a aquello que pretenda hacer, pues también permitirá utilizar todos las nuevas habilidades adquiridas en el desarrollo del mismo.

Por último, y en relación al anterior punto, influye la motivación personal. Ningún TFG es sencillo, y por eso es normal que en ciertas ocasiones se pueda perder la motivación de realizarlo. Escoger un tema familiar y que resulte entretenido puede ser clave para mantener un flujo de trabajo constante y cumplir con los objetivos del TFG, tanto temporales como en cuanto a características técnicas.

Por todas estas razones considero que el tema elegido es perfecto para mi situación y necesidades personales.

1.2. Siglas y definiciones importantes

- **MOBA:** Un videojuego MOBA, siglas de *Multiplayer Online Battle Arena* o Arena de Batalla Multijugador Online, es un tipo de videojuego de estrategia y acción generalmente jugado por dos equipos de unos 3 a 5 jugadores. El objetivo es alcanzar o destruir un objetivo enemigo antes de que el equipo contrario destruya el propio. Este objetivo estará defendido por distintas estructuras y defensas que impedirán el paso al equipo atacante y deberán ser destruídas antes de poder llegar a dicho objetivo. En el proceso, ambos equipos luchan a lo largo del mapa para ir avanzando, a la vez que se ralentizan el avance enemigo. Normalmente se dispone de una gran variedad de personajes para elegir al comienzo de la partida, cada uno con habilidades únicas.

- **Top-Down:** Cuando se habla sobre videojuegos, top-down hace referencia a su modo de visualización, viéndose el mapa desde la parte superior de este, como si de una cámara aérea se tratase. Del mismo modo, los personajes son visualizados y controlados desde arriba, generalmente moviéndose por un entorno en 2 dimensiones.

- **GUI:** Siglas correspondientes a *Graphical User Interface*. Hace referencia a toda interfaz que haga uso de diferentes imágenes, texto y objetos en pantalla para representar la información que se ofrece al usuario de un sistema y proporcionarle diferentes opciones de interacción con éste.

- **Pixel-art:** Se califica como Pixel-art a la creación y edición de imágenes a nivel de píxel, es decir, cada uno de los píxeles de éstas son editados y colocados de forma individual. Usualmente las imágenes generadas son de baja resolución. Es un formato muy popular en el desarrollo de videojuegos, pues otorga un efecto retro a los juegos que utilizan este estilo.

- **Tile / Tileset:** Un tile es una pequeña imagen, generalmente cuadrada, que mediante la combinación con otros tiles se utiliza para generar escenarios y fondos de pantalla en videojuegos. Estos tiles se recogen en ficheros que los agrupan, conocidos como tilesets. La ventaja de la utilización de tiles frente a imágenes individuales es que mediante las diferentes combinaciones entre los tiles de un tileset se pueden conseguir una gran

cantidad de imágenes diferentes, lo cual simplifica la edición y reduce la carga del diseño en aspectos gráficos.

- **Lobby:** En un videojuego online se conoce como lobby a la pantalla que aparece de forma previa a que el juego comience, siendo esta en la que generalmente se pueden formar los equipos, escoger personajes, y todas las preparaciones necesarias para comenzar la partida.

2. Planteamiento Inicial

En esta sección se trata la definición de los objetivos y el alcance de este proyecto, así como la planificación temporal de las tareas definidas junto a una evaluación económica general. También se incluyen los posibles riesgos a tener en cuenta durante el desarrollo del proyecto, y se discuten las posibles herramientas a utilizar para la realización del mismo.

2.1. Objetivos

Para el completo desarrollo de este proyecto se proponen a continuación unos objetivos básicos a cumplir:

- Desarrollo de la base para un juego funcional y con todas las características básicas de un MOBA, así como con los cambios propuestos para que sea diferente al resto de juegos disponibles.
- Implementación total del modo multijugador, esencial para un videojuego MOBA. Este modo debe ser totalmente transparente de cara al usuario, de modo que no necesite ningún tipo de configuración especial por su parte para poder utilizarlo.
- Creación de un juego multiplataforma. Es imprescindible que el juego pueda ejecutarse sin problemas en los principales sistemas operativos (Windows, Linux y Mac).
- Profundización y aprendizaje en las técnicas necesarias para el desarrollo de videojuegos de una forma profesional, por razones tanto académicas como por mera satisfacción personal de ver los resultados de un trabajo bien hecho.

2.2. Alcance

En este apartado se tratará todo aquello relacionado con cómo se va a desarrollar el proyecto, incluyendo la elección de un ciclo de vida adecuado y su justificación, así como la correspondiente descomposición del trabajo a realizar.

2.2.1. Ciclo de vida

Este proyecto se desarrollará según el modelo de ciclo de vida incremental. Este ciclo de vida consiste en la elaboración de una serie de prototipos, basándose cada uno en el anterior y añadiendo nuevas funcionalidades hasta alcanzar un estado final en el que se hayan satisfecho los objetivos del proyecto. A su vez, el desarrollo de cada prototipo seguirá un ciclo de vida

en cascada, debiendo realizar para cada uno de ellos una implementación y pruebas propias. En la figura 1 se puede observar el funcionamiento de este modelo de forma gráfica.

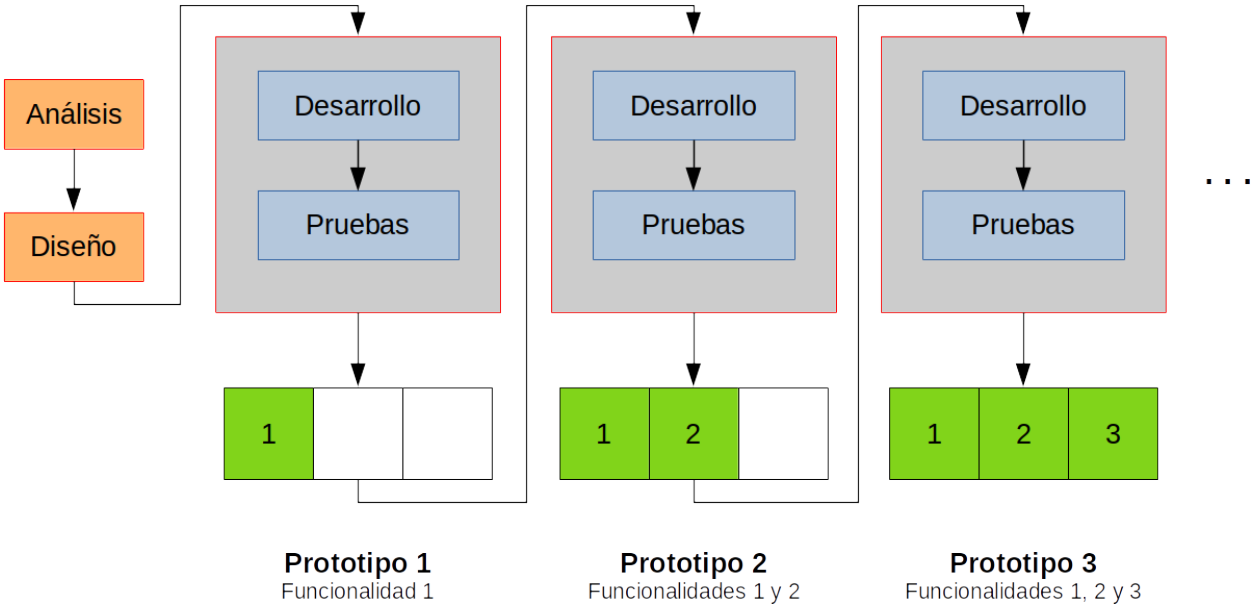


Figura 1: Ciclo de vida incremental

Se ha decidido utilizar este modelo por diversas razones. En primer lugar, es muy útil disponer de prototipos intermedios, pues gracias a ellos se pueden comprobar que los objetivos se van cumpliendo de forma progresiva. Además, siendo este un proyecto grande, es beneficioso poder hacer tests de forma incremental, puesto que es preferible solucionar errores en su etapa más temprana, pues siguiendo este modelo no se podrá pasar de un prototipo al siguiente mientras tenga fallos, por lo que nos aseguramos que cualquier fallo que se encuentre es debido al añadido de la última iteración. Por otro lado, durante el desarrollo de un videojuego es complicado implementar funcionalidades sin un contexto mínimo: por ejemplo, no tiene sentido implementar un personaje jugable antes de tener disponible un escenario donde ubicarlo. Es por esto que es muy necesario dividir la implementación en prototipos, pues implementarlas por separado y testear todas estas funcionalidades a la vez crea un alto riesgo de errores que podrían tener soluciones difíciles.

Además de lo anterior, en la figura 1 se puede observar que tanto el análisis como el diseño están ubicados fuera del ciclo de cada prototipo. Al utilizar la metodología del ciclo de vida incremental generalmente se ubican dentro de estos ciclos, y es necesario realizar un análisis y diseño específico para cada prototipo. Sin embargo, para este proyecto es beneficioso realizar un análisis y diseño global, por razones similares a las que se han expuesto anteriormente: muchas de las funcionalidades son altamente dependientes de otras y es preferible tener una vision general del diseño del proyecto en conjunto para poder implementar en consecuencia a éste.

Para este proyecto se realizarán los siguientes prototipos:

1. **Base de datos:** Incluirá la base de datos y el servidor mediante el cual se harán las conexiones a esta.
2. **Menús, login y registro:** Se implementará toda la funcionalidad perteneciente a los menús de la aplicación, incluido el login y el registro.
3. **Creación de escenario y personajes jugables:** Se creará el escenario del juego, así como el acceso al mismo a través de los menús. Se implementará toda la funcionalidad relativa a los personajes jugables y la interacción de los jugadores con ellos por medio de los controles. Además, estos personajes han de ser capaces de interactuar con el entorno del escenario.
4. **Control del estado de la partida:** Se añadirá el control del flujo de eventos de la partida, es decir, cómo de cerca está cada equipo de la victoria, sus estadísticas, etc. También se controlará en qué momento un equipo gana y se actuará de forma acorde, guardando los resultados en la base de datos.
5. **Diseño final de gráficos:** Se añadirá a todos los jugadores y entidades los gráficos definitivos con sus animaciones correspondientes.

2.2.2. Estructura de descomposición del trabajo

Para conseguir una organización útil de las tareas (también llamadas paquetes de trabajo) en las que se dividirá el proyecto, estas han sido agrupadas en diferentes módulos en función de las necesidades del proyecto que abarquen o que intenten solventar. La distribución de dichas tareas se puede observar de forma gráfica en la figura 2:

Tal y como está representado en el anterior diagrama, los módulos definidos son los siguientes:

- **Documentación:** En este módulo contiene todas las tareas relativas a la documentación por escrito del proyecto para su posterior entrega y presentación.
- **Gestión:** Aquí entran todos los paquetes de trabajo relacionados con la definición de los objetivos y el alcance del proyecto, así como los que traten acerca de la organización y planificación del proyecto.
- **Aprendizaje:** Se incluyen aquellas tareas necesarias para la preparación de las herramientas utilizadas en la realización del proyecto, su instalación y la búsqueda de información acerca de su funcionamiento, incluyendo todo el tiempo necesario para su entendimiento y aprendizaje en caso de que haga falta.

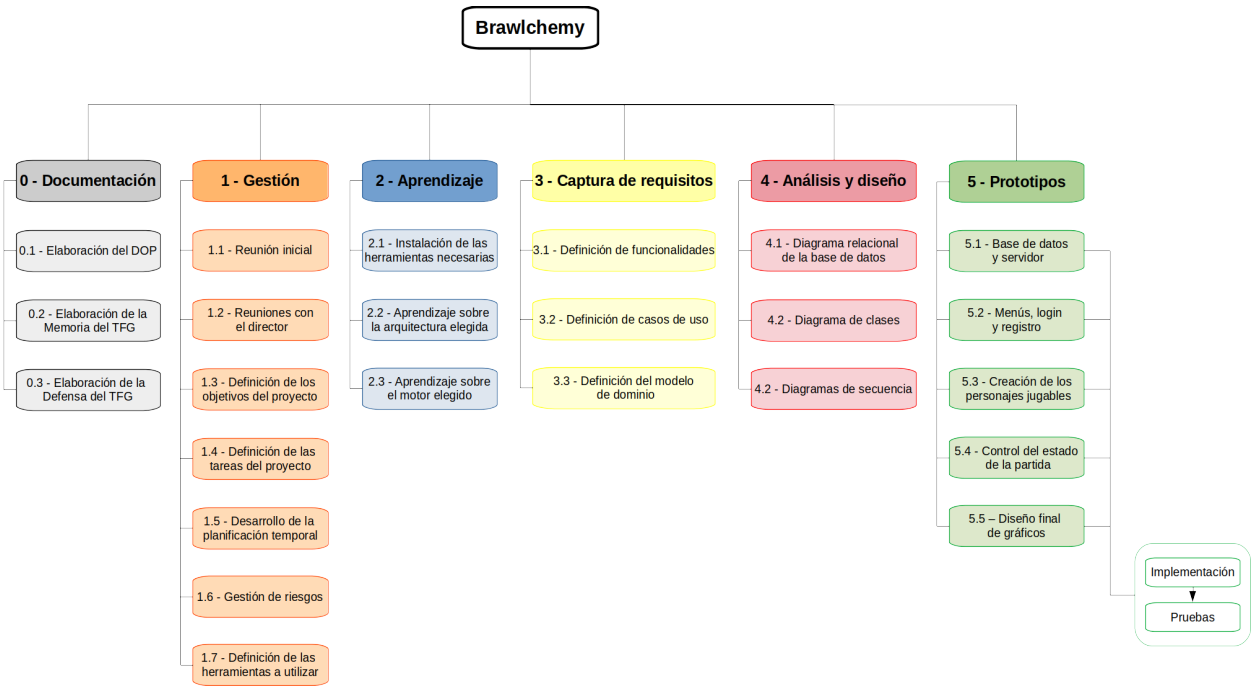


Figura 2: Diagrama EDT

- **Captura de requisitos:** Dentro de este módulo están las tareas que abarquen la definición de los requerimientos y funcionalidades generales del proyecto y aquellas específicas para los usuarios.
- **Análisis y diseño:** Aquí se encuentran los paquetes de trabajo que dividirán la solución en partes manejables y definirán la estructura final del código del proyecto, todo esto de forma documentada mediante explicaciones exhaustivas sobre cada decisión y diagramas en caso de ser posible, y que servirán como base para el posterior desarrollo software.
- **Prototipos:** Este módulo contiene el desarrollo de los prototipos propuestos para la realización del proyecto. Como se puede observar en la figura 2, cada prototipo ha sido representado a su vez con una fase de desarrollo y una de pruebas específica, siendo estas realizadas para cada uno de ellos por los motivos explicados en el apartado Ciclo de vida.

Los paquetes de trabajo que se incluyen en cada módulo se detallan a fondo en las siguientes tablas:

Tabla 1: Elaboración del DOP

Módulo 0: Documentación **Paquete de trabajo 0.1:** Elaboración del DOP
Duración: 30 horas.
Descripción: Elaboración del Documento de Objetivos del Proyecto, incluyendo éste objetivos, planificación, evaluación económica y gestión de riesgos del proyecto.
Precedencias: No.
Salidas: Documento de Objetivos del Proyecto.

Tabla 2: Elaboración de la Memoria del TFG

Módulo 0: Documentación **Paquete de trabajo 0.2:** Elaboración de la Memoria del TFG
Duración: 40 horas.
Descripción: Elaboración de la Memoria del Trabajo de Fin de Grado, consistente en la información relativa al diseño del juego y la documentación del desarrollo y pruebas de los diferentes prototipos.
Precedencias: No: trabajo continuo.
Salidas: Memoria del Trabajo de Fin de Grado.

Tabla 3: Elaboración de la Defensa del TFG

Módulo 0: Documentación **Paquete de trabajo 0.3:** Elaboración de la Defensa del TFG
Duración: 20 horas.
Descripción: Elaboración y preparación de la Defensa del Proyecto y el material de apoyo necesario para ésta.
Precedencias: Módulos 1, 2, 3, 4 y 5. Paquetes de trabajo 0.1 y 0.2.
Salidas: Material para la Defensa del Proyecto.

Tabla 4: Reunión inicial

Módulo 1: Gestión **Paquete de trabajo 1.1:** Reunión inicial
Duración: 1 hora.
Descripción: Reunión inicial en la que se discuten las posibles ideas con el director y se da el visto bueno a la propuesta final del TFG.
Precedencias: No.
Salidas: Documento de la Propuesta del Estudiante.

Tabla 5: Reuniones con el director

Módulo 1: Gestión **Paquete de trabajo 1.2:** Reuniones con el director
Duración: 5 horas.
Descripción: Reuniones periódicas con el director del proyecto en las que se harán controles del avance del proyecto o se resolverán dudas que hayan podido surgir de cara a la realización de del proyecto o su documentación.
Precedencias: Paquete de trabajo 1.1.
Salidas: Documento consistente en un breve resumen de los aspectos tratados en cada reunión.

Tabla 6: Definición de los objetivos del proyecto

Módulo 1: Gestión **Paquete de trabajo 1.3:** Definición de los objetivos del proyecto
Duración: 2 horas.
Descripción: Establecer y describir los objetivos del proyecto, haciendo distinción entre aquellos que sean imprescindibles y aquellos que se consideren secundarios u opcionales.
Precedencias: Paquete de trabajo 1.1.
Salidas: Documento que recoja los objetivos principales y secundarios del proyecto.

Tabla 7: Definición de las tareas del proyecto

Módulo 1: Gestión **Paquete de trabajo 1.4:** Definición de las tareas del proyecto
Duración: 6 horas.
Descripción: Establecer las tareas necesarias para la elaboración del proyecto y el cumplimiento de los objetivos, así como un modelo de ciclo de vida adecuado para éstas.
Precedencias: Paquete de trabajo 1.3.
Salidas: Documento que recoja de forma explicada las tareas del proyecto y el ciclo de vida elegido.

Tabla 8: Desarrollo de la planificación temporal y económica

Módulo 1: Gestión **Paquete de trabajo 1.5:** Desarrollo de la planificación temporal y económica
Duración: 4 horas.
Descripción: Establecer un orden para las tareas, definiendo para cada una sus precedencias y una estimación del tiempo necesario para su realización. Cálculo del impacto económico del proyecto en conjunto.
Precedencias: Paquete de trabajo 1.4.
Salidas: Diagrama Gantt correspondiente a la planificación temporal. Evaluación económica del proyecto.

Tabla 9: Gestión de riesgos

Módulo 1: Gestión **Paquete de trabajo 1.6:** Gestión de riesgos
Duración: 5 horas.
Descripción: Definición de los riesgos que afectan al proyecto, junto con sus correspondientes planes de prevención y contingencia.
Precedencias: Paquete de trabajo 1.4.
Salidas: Documento explicativo en el que se incluya toda la información sobre los riesgos, los planes de prevención y los planes de contingencia.

Tabla 10: Definición de las herramientas a utilizar

Módulo 1: Gestión **Paquete de trabajo 1.7:** Definición de las herramientas a utilizar
Duración: 3 horas.
Descripción: Comparación entre las distintas herramientas disponibles para la realización del proyecto, desde el ámbito de la documentación hasta el del desarrollo software.
Precedencias: Paquete de trabajo 1.3.
Salidas: Documento explicativo en el que se incluyan las diferentes razones detrás de las elecciones de herramientas a utilizar.

Tabla 11: Instalación de las herramientas necesarias

Módulo 2: Aprendizaje **Paquete de trabajo 2.1:** Instalación de las herramientas necesarias
Duración: 3 horas.
Descripción: Instalación de las diferentes herramientas que vayan a ser utilizadas en el proyecto junto con su correspondiente configuración, puesta a punto y pruebas necesarias para comprobar su funcionamiento.
Precedencias: Paquete de trabajo 1.6.
Salidas: Herramientas preparadas para su uso.

Tabla 12: Aprendizaje sobre la arquitectura elegida

Módulo 2: Aprendizaje **Paquete de trabajo 2.2:** Aprendizaje sobre la arquitectura elegida
Duración: 25 horas.
Descripción: Búsqueda, lectura y aprendizaje necesario de la documentación de las herramientas y tecnologías seleccionadas para la realización de la arquitectura del proyecto, y realización de diferentes pruebas para comprobar y reforzar el aprendizaje de esta información.
Precedencias: Paquete de trabajo 2.1.
Salidas: Documento que incluya todas las fuentes útiles de documentación para poder disponer de ellas durante el desarrollo.

Tabla 13: Aprendizaje sobre el motor elegido

Módulo 2: Aprendizaje **Paquete de trabajo 2.3:** Aprendizaje sobre el motor elegido
Duración: 15 horas.
Descripción: Búsqueda, lectura y aprendizaje necesario de la documentación de las herramientas y tecnologías seleccionadas para la realización del juego, y realización de diferentes pruebas y demos para comprobar y reforzar el aprendizaje de esta información.
Precedencias: Paquete de trabajo 2.1.
Salidas: Documento que incluya todas las fuentes útiles de documentación para poder disponer de ellas durante el desarrollo.

Tabla 14: Definición de funcionalidades

Módulo 3: Captura de Requisitos **Paquete de trabajo 3.1:** Definición de funcionalidades
Duración: 3 horas.
Descripción: Definir las funcionalidades necesarias del proyecto para cumplir con los objetivos de éste.
Precedencias: Paquete de trabajo 1.4.
Salidas: Documento que incluya las funcionalidades generales del proyecto, así como aquellas asignadas a cada uno de los prototipos.

Tabla 15: Definición de casos de uso

Módulo 3: Captura de Requisitos **Paquete de trabajo 3.2:** Definición de casos de uso
Duración: 15 horas.
Descripción: Especificación de los casos de uso de los usuarios del sistema, representando las funcionalidades del sistema junto al usuario que las lleva a cabo.
Precedencias: Paquete de trabajo 3.1.
Salidas: Diagrama general de casos de uso y casos de uso extendidos.

Tabla 16: Definición del modelo de dominio

Módulo 3: Captura de Requisitos **Paquete de trabajo 3.3:** Definición del modelo de dominio
Duración: 1 hora.
Descripción: Especificación de los datos y entidades que se van a almacenar en forma de diagrama, representando éste las relaciones entre dichas entidades.
Precedencias: Paquete de trabajo 3.1.
Salidas: Diagrama de modelo de dominio.

Tabla 17: Diagrama relacional de la base de datos

Módulo 4: Análisis y Diseño **Paquete de trabajo 4.1:** Diagrama relacional de la base de datos
Duración: 2 horas.
Descripción: Transformación del modelo de dominio en un diagrama relacional de la base de datos.
Precedencias: Paquete de trabajo 3.3.
Salidas: Diagrama relacional de la base de datos.

Tabla 18: Diagrama de clases

Módulo 4: Análisis y Diseño **Paquete de trabajo 4.2:** Diagrama de clases
Duración: 15 horas.
Descripción: Desarrollo del diagrama de clases del juego, en el que se incluirán las diferentes clases del proyecto con sus métodos más representativos.
Precedencias: Módulo 3 y Paquete de Trabajo 4.1
Salidas: Diagrama de clases del juego.

Tabla 19: Diagramas de secuencia

Módulo 4: Análisis y Diseño **Paquete de trabajo 4.3:** Diagramas de secuencia
Duración: 20 horas.
Descripción: Diagramas de secuencia de las principales funcionalidades del juego, en los que se representará la interacción secuencial entre las clases y sus llamadas.
Precedencias: Paquete de trabajo 4.2.
Salidas: Diagramas de secuencia.

Tabla 20: Base de datos y servidor

Módulo 5: Prototipos **Paquete de trabajo 5.1:** Base de datos y servidor
Duración: 15 horas.
Descripción: Desarrollo del prototipo 1, en el que implementará la base de datos y el servidor en el que se alojará, así como las conexiones que se realizaran a ésta desde el juego. Incluye la fase de desarrollo y la fase de pruebas.
Precedencias: Módulo 4.
Salidas: Prototipo 1.

Tabla 21: Menús, login y registro

Módulo 5: Prototipos **Paquete de trabajo 5.2:** Menús, login y registro
Duración: 15 horas.
Descripción: Desarrollo del prototipo 2, en el que se implementarán los menús del juego, así como la funcionalidad que permite registrarse e iniciar sesión en el sistema. Incluye la fase de desarrollo y la fase de pruebas.
Precedencias: Paquete de trabajo 5.1 (Prototipo 1).
Salidas: Prototipo 2.

Tabla 22: Creación de los personajes jugables

Módulo 5: Prototipos **Paquete de trabajo 5.3:** Creación de los personajes jugables
Duración: 25 horas.
Descripción: Desarrollo del prototipo 3, el que se implementarán los personajes jugables y el entorno en el que se ubicarán, junto a las interacciones entre ellos. Incluye la fase de desarrollo y la fase de pruebas.
Precedencias: Paquete de trabajo 5.2 (Prototipo 2).
Salidas: Prototipo 3.

Tabla 23: Control del estado de la partida

Módulo 5: Prototipos **Paquete de trabajo 5.4:** Control del estado de la partida
Duración: 25 horas.
Descripción: Desarrollo del prototipo 4, en el que se implementará el control del estado de la partida, tanto el inicio y el final de ésta, como el almacenamiento de los resultados. Incluye la fase de desarrollo y la fase de pruebas.
Precedencias: Paquete de trabajo 5.3 (Prototipo 3).
Salidas: Prototipo 4.

Tabla 24: Diseño final de gráficos

Módulo 5: Prototipos **Paquete de trabajo 5.5:** Diseño final de gráficos
Duración: 20 horas.
Descripción: Desarrollo del prototipo 5, en el que se implementarán los gráficos finales del juego para personajes, entorno y GUI. Incluye la fase de desarrollo y la fase de pruebas.
Precedencias: Paquete de trabajo 5.4 (Prototipo 4).
Salidas: Prototipo 5.

2.3. Planificación temporal

Después de haber definido las tareas del proyecto, y sabiendo la duración estimada de éstas, debemos distribuirlas a lo largo del tiempo para conocer el momento en que cada una puede comenzar, y como consecuencia la duración total del proyecto. Para una mejor visualización de esta distribución se utilizará un diagrama Gantt en el que las tareas aparecerán según su duración en horas.

Sin embargo, pese a que ciertas tareas puedan ser realizadas de forma paralela, esto no será posible en la realización de este proyecto, puesto que sólo se dispone de una persona. Dos tareas que se pudiesen realizar en paralelo se representarán una detrás de la otra. Es por eso que tareas que no tienen una precedencia directa con otra diferente puede que aparezcan como tal en el diagrama. Sin embargo y, como es obvio, toda tarea aparecerá siempre más tarde que la precedente, aunque no sea inmediatamente después.

Por otro lado, ciertas tareas tienen un peso en horas definido, pero serán, en cierta medida, tareas continuas. Un ejemplo de esto es la realización de la memoria, siendo una tarea que se realizará de forma continua junto al resto de tareas. De forma similar, tareas recurrentes a lo largo del tiempo pero sin fecha definida, como las reuniones con el director, se realizarán en momentos puntuales a lo largo del desarrollo del proyecto. En estos dos casos se representarán en el diagrama como tareas «largas» y con un color más oscuro que el resto de las de su módulo, indicando que su representación no se corresponde a su duración en horas sino a su extensión a lo largo del tiempo.

El diagrama resultante se puede visualizar en la figura 3:

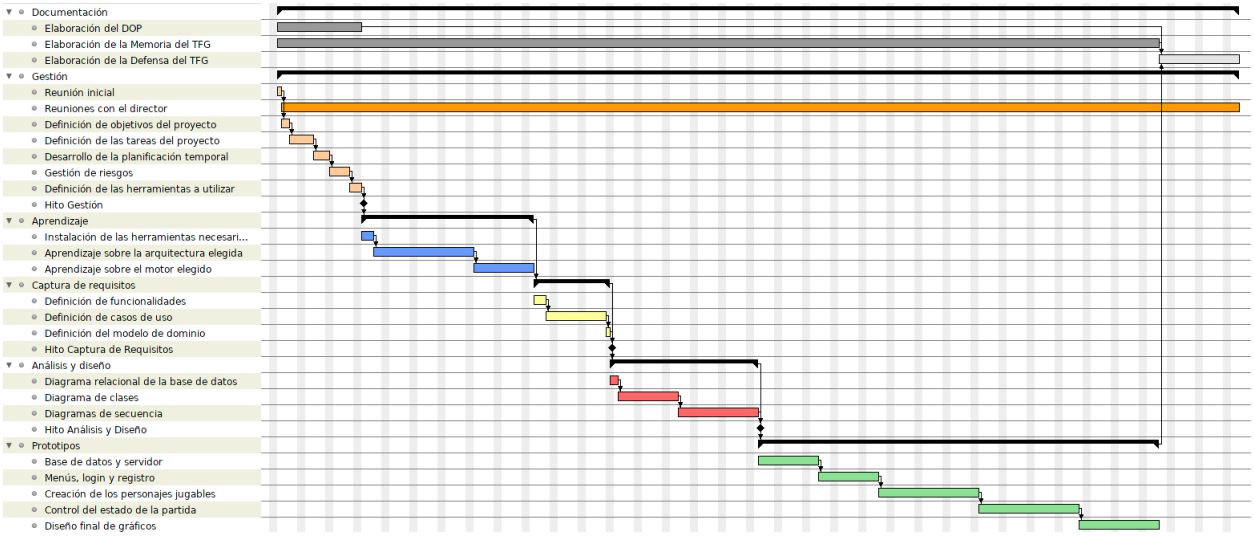


Figura 3: Diagrama Gantt

En la tabla 25 se calculan las horas parciales y totales de cada módulo y del proyecto completo:

Tabla 25: Duración total del proyecto

Tarea	**Duración**
Documentación	**90**
Elaboración del DOP	30
Elaboración de la Memoria del TFG	40
Elaboración de la Defensa del TFG	20
Gestión	**26**
Reunión inicial	1
Reuniones con el director	5
Definición de los objetivos del proyecto	2
Definición de las tareas del proyecto	6
Desarrollo de la planificación temporal y económica	4
Gestión de riesgos	5
Definición de las herramientas a utilizar	3
Aprendizaje	**43**
Instalación de las herramientas necesarias	3
Aprendizaje sobre la arquitectura elegida	25
Aprendizaje sobre el motor elegido	15
Captura de Requisitos	**19**
Definición de funcionalidades	3
Definición de casos de uso	15
Definición del modelo de dominio	1
Análisis y diseño	**37**
Diagrama relacional de la base de datos	2
Diagrama de clases	15
Diagramas de secuencia	20
Prototipos	**100**
Base de datos y servidor	15
Menús, login y registro	15
Creación de los personajes jugables	25
Control del estado de la partida	25
Diseño final de gráficos	20
TOTAL	**315**

Es necesario también definir la carga de trabajo diaria. Para la realización de este proyecto se supondrá que para los días laborables (lunes a jueves para el desarrollador) la carga de trabajo será de unas 3 horas. Por otro lado, para el conjunto de los días no laborables (viernes a domingo para el desarrollador) se estima una carga total de unas 13 horas. Según esto cada semana tendrá una carga de trabajo aproximada de 25 horas.

Teniendo en cuenta la duración del proyecto y la carga de trabajo semanal, podemos hacer una estimación de las semanas que durará el proyecto. Para simplificar, se considerará la carga de trabajo de 25 horas semanales para todas las semanas, no teniendo en cuenta posibles días festivos.

Las semanas necesarias para la realización del proyecto desglosadas por módulos se muestran en la tabla 26. Los días han sido redondeados en caso de ser necesario.

Tabla 26: Duración total del proyecto en semanas

Tarea	**Duración**
Documentación	3 semanas y 4 días
Gestión	1 semana
Aprendizaje	1 semanas y 5 días
Captura de Requisitos	5 días
Análisis y diseño	1 semana y 4 días
Prototipos	4 semanas
TOTAL	**12 semanas y 4 días**

2.4. Herramientas

En esta sección se detallan las herramientas y programas que se utilizarán para la realización del proyecto y su documentación, así como una explicación de su utilidad y la razón de su elección.

- **Git:** Git es una herramienta de control de versiones de código abierto. Es la que se utilizará en este proyecto, principalmente debido a su popularidad y al conocimiento previo sobre su modo de uso, y servirá para disponer de las distintas versiones a lo largo del tiempo, especialmente útil para la finalización de cada uno de los prototipos.
- **GitLab:** GitLab es la plataforma elegida para alojar las versiones del proyecto en sus

repositorios utilizando Git. Dispone de todas las funcionalidades necesarias de forma gratuita, entre ellas la posibilidad de crear repositorios privados.

- **Google Cloud:** Google Cloud dispone de Compute Engine, un servicio que proporciona máquinas virtuales en la nube a las que se puede acceder a través de internet. Será en una de estas máquinas en la que se implementará el servidor de base de datos y las instancias de los servidores de Brawlchemy.

- **MySQL:** MySQL es el sistema de gestión de bases de datos elegido para el desarrollo de este proyecto, debido a su alto grado de compatibilidad con otras aplicaciones y lenguajes, así como por la familiaridad con su lenguaje y la calidad y cantidad de documentación de la que se dispone.

- **NodeJS:** NodeJS es un lenguaje de programación de servidores basado en JavaScript que se utilizará para el desarrollo de la interacción de Brawlchemy con la base de datos. Es especialmente indicado por la facilidad de conectarlo con una base de datos MySQL, así como por la seguridad que ofrece de forma muy sencilla. Además, dispone de una gran cantidad de documentación debido al auge en que se encuentra actualmente.

- **Postman:** Postman es un programa totalmente gratuito que sirve para realizar diferentes peticiones HTTP y analizar los resultados de estas, pudiendo así automatizar el testeo de servicios web en desarrollo. Este programa será utilizado para comprobar el correcto funcionamiento del servidor creado en NodeJS mediante el que nos conectaremos al servidor de Brawlchemy.

- **LaTeX:** LaTeX es un sistema de composición de textos especialmente indicado para el desarrollo de documentos técnicos. Este se utilizará para generar la documentación del proyecto, siendo elegido frente a otras alternativas por la organización que ofrece de cara a una documentación grande y la facilidad para generar documentos limpios y profesionales.

- **Atom:** Atom es un editor de textos libre desarrollado por GitHub. Es una herramienta muy útil para éste proyecto puesto que permite de forma nativa la edición de una gran variedad de lenguajes, además de poseer una gran biblioteca de plugins que podrán hacer su uso incluso más sencillo y adecuado. Este editor es la perfecta elección para la edición del código LaTeX que formará la documentación, y también se utilizará para el desarrollo de código JavaScript que será ejecutado en el servidor.

- **PlantUML:** PlantUML es una librería que permite la creación de distintos tipos de diagramas a partir de texto plano. Debido a su facilidad de uso, a la ligereza (pues no necesita ningún tipo de programa aparte de un editor de texto) y a la compatibilidad con Atom, será el método utilizado para la creación de diagramas de casos de uso, de modelo de dominio, de clases y de secuencia.

- **GanttProject:** GanttProject es una herramienta de desarrollo de diagramas Gantt de forma sencilla. Incluye todas las herramientas necesarias para crear los diagramas que representarán la planificación del proyecto.

- **LibreOffice Draw:** LibreOffice Draw es una de las funcionalidades incluidas en el paquete de LibreOffice, siendo utilizada para la creación de diagramas y gráficos simples. En el contexto del proyecto, se utilizará para el diseño de diagramas o esquemas simples que no dispongan de una herramienta especializada.

- **Aseprite:** Aseprite es una potente herramienta de edición de gráficos especialmente indicada para el diseño de imágenes pixel-art y gráficos de videojuegos. Entre sus ventajas encontramos la simplicidad de uso, el gran número de funcionalidades optimizadas para tilesets, y la posibilidad de diseñar y exportar animaciones.

En cuanto a los motores gráficos, la elección se ha realizado entre los siguientes:

- **Unity:** Unity es uno de los motores gráficos más populares, debido principalmente a su potencia, gran abanico de funcionalidades y amplia variedad de extensiones y librerías. Es un motor orientado y optimizado hacia el diseño 3D, aunque dispone de editor de juegos en 2D. El lenguaje de programación utilizado en los scripts de Unity es C#. Dispone de una versión gratuita con licencia de uso personal e incluyendo prácticamente todas las funcionalidades del motor, así como versiones completas con precios desde 25$/mes.

- **GameMaker Studio 2:** GameMaker Studio es un motor especialmente orientado a la programación de videojuegos para usuarios sin un gran conocimiento de lenguajes de programación, pues utiliza un sistema de *drag and drop* (arrastrar y soltar) que permite la edición mediante la organización de diferentes iconos y ventanas en la pantalla. Dispone de una versión gratuita con funcionalidad limitada y una versión completa desde 39$/año.

- **Godot 3:** Godot 3 es un motor gráfico de código abierto diseñado para el desarrollo de juegos tanto en 2 como en 3 dimensiones. Dispone de dos versiones principales, una que utiliza el lenguaje de programación GDScript (muy similar en sintaxis a Python) y otra que utiliza C#, además de versiones especialmente diseñadas y optimizadas para ser utilizadas en servidores. Es un motor totalmente gratuito.

- **Cocos2d:** Cocos2d es un motor de código abierto orientado al desarrollo de juegos en 2 dimensiones. Es un motor muy ligero, pues dispone únicamente de las funcionalidades básicas para diseñar juegos 2D. Cabe destacar que dispone de diferentes versiones que soportan nativamente el desarrollo en diferentes lenguajes de programación, entre ellos Python, JavaScript, C++ o C#. Esto lo hace indicado para adaptarse a los conocimientos previos de los desarrolladores. Es un programa completamente gratuito.

La decisión entre uno de estos motores para el desarrollo de Brawlchemy se tomará después de analizar las alternativas de juegos existentes en el mercado al final del apartado Antecedentes, donde se justificará de forma extensa los motivos detrás de dicha decisión.

2.5. Gestión de riesgos

Es posible que durante el desarrollo de cualquier proyecto surjan situaciones que afecten negativamente al mismo. Estos problemas pueden aumentar la carga de trabajo de los desarrolladores, hacer que se pierda tiempo y dinero y, en última instancia y en situaciones más graves, pueden hacer que todo el proyecto tenga que ser cancelado o reiniciado completamente. Es por esto que es necesario crear un plan de riesgos en el que se detecten los posibles riesgos que puedan afectar al proyecto, se identifiquen sus causas y consecuencias, y se defina un plan de acción tanto para prevenir la situación (reduciendo al mínimo la probabilidad de que ocurra), como para mitigar su impacto (reduciendo al mínimo el efecto que tenga en caso de ocurrir).

Antes de definir los riesgos, es conveniente definir de forma aproximada cuál es la probabilidad de que ocurra. Sin embargo, es difícil asignar valores exactos a estas probabilidades. Para solucionarlo se definirán unos rangos de probabilidades, siendo estos los que se usarán para valorar los riesgos. Estos se pueden ver en la tabla 27:

Tabla 27: Probabilidades de los diferentes riesgos

Tipo de riesgo	**Probabilidad**
Improbable	0 % - 5 %
Poco probable	5 % - 50 %
Probable	50 % - 75 %
Muy probable	75 % - 100 %

De una forma similar a la anterior, cada riesgo puede tener un efecto asociado diferente. Generalmente se mide este retraso en horas de trabajo necesarias para paliar los efectos. Ya que, de igual forma que con las probabilidades, es difícil establecer una cantidad exacta, se utilizarán los rangos definidos en la tabla 28:

Para cada riesgo, aparte de su probabilidad y su impacto, se deberá definir un plan de prevención y un plan de contingencia. El plan de prevención trata de reducir al mínimo la probabilidad de que el riesgo ocurra. El plan de contingencia se pone en marcha cuando el riesgo se ha dado, y engloba todas las medidas necesarias para mitigar las consecuencias de este. Además, es necesario establecer revisiones periódicas para asegurar que sendos planes se están cumpliendo.

Tabla 28: Impacto de los diferentes riesgos

Impacto	**Retraso asociado**
Leve	Menor que 1 día
Medio	1 - 2 días
Alto	2 - 7 días
Muy alto	Mayor que 7 días

A continuación se listarán los riesgos detectados, divididos en tres categorías: riesgos hardware y software, provocados por problemas en los equipos de trabajo; riesgos de diseño, causados por errores o modificaciones en la captura de requisitos, el diseño o la planificación del proyecto; y riesgos externos, siendo estos los relacionados con problemas que puedan afectar personalmente al desarrollador pero cuyas causas no tengan que ver con el propio proyecto.

2.5.1. Riesgos hardware y software

Tabla 29: Riesgo de infección por virus informático

Riesgo de infección por virus informático	
Descripción	Una infección por virus informático a uno de los equipos del proyecto podría causar una perdida de archivos, inutilizar el equipo durante un tiempo o, en un caso crítico, para siempre.
Probabilidad	Improbable.
Impacto	Muy alto.
Plan de prevención	Utilizar sistemas Unix para minimizar el riesgo de virus. Evitar las descargas de herramientas desde sitios web no oficiales. Realización de copias de seguridad del sistema de forma periódica.
Plan de contingencia	Reparación del equipo por un profesional en caso de ser posible o necesario. Restauración de las copias de seguridad en un nuevo equipo en caso contrario.

Tabla 30: Riesgo de avería del equipo

Riesgo de avería del equipo	
Descripción	Una avería en uno de los equipos podría inutilizarlo durante un tiempo, dando lugar a un retraso en el desarrollo del proyecto.
Probabilidad	Improbable.
Impacto	Alto.
Plan de prevención	Realización de copias de seguridad del sistema y del proyecto de forma periódica para que estas puedan ser restauradas en caso de ser necesario.
Plan de contingencia	Reparación del equipo por un profesional en caso de ser posible o necesario. Restauración de las copias de seguridad en un nuevo equipo en caso contrario.

Tabla 31: Riesgo de cambios críticos en el software utilizado

Riesgo de cambios críticos en el software utilizado	
Descripción	Una modificación o actualización crítica en una herramienta o librería utilizada podría inutilizar gran parte del código desarrollado en base a su versión previa, dejándolo instantáneamente obsoleto.
Probabilidad	Improbable.
Impacto	Muy alto.
Plan de prevención	Comprobar la versión actual de todas las herramientas que se utilicen, y determinar de forma previa a la implementación si alguna de las herramientas necesarias prevé la realización de cambios importantes.
Plan de contingencia	Estudio a fondo de la documentación relativa a los cambios y adaptación del código a la nueva versión.

Tabla 32: Riesgo de perdida de datos

Riesgo de perdida de datos	
Descripción	Existe la posibilidad de que se pierdan datos del proyecto debido a actualizaciones, problemas con alguno de los programas utilizados o simplemente por un error humano.
Probabilidad	Probable.
Impacto	Muy alto.
Plan de prevención	Realización de copias de seguridad del sistema y del proyecto de forma periódica para que estas puedan ser restauradas en caso de ser necesario.
Plan de contingencia	Restauración de las copias de seguridad más recientes.

2.5.2. Riesgos de diseño

Tabla 33: Riesgo de planificación errónea

Riesgo de planificación errónea	
Descripción	En caso de desarrollar una planificación temporal incorrecta, el proyecto puede verse retrasado respecto a los tiempos definidos.
Probabilidad	Probable.
Impacto	Medio.
Plan de prevención	Realizar una planificación que esté lo más adaptada a los tiempos reales posible. En caso de duda, será preferible redondear las duraciones estimadas al alza.
Plan de contingencia	Modificar la planificación temporal añadiendo los retrasos y reubicando las tareas restantes.

Tabla 34: Riesgo de análisis y diseño erróneo

Riesgo de análisis y diseño erróneo	
Descripción	En caso de no definir unos diagramas consistentes, claros y completos en base a las funcionalidades, es posible que durante la implementación sea necesario modificarlos para que estos reflejen correctamente todas las funcionalidades del proyecto.
Probabilidad	Probable.
Impacto	Medio.
Plan de prevención	Realizar un diseño modular y claro que haga sencillo encontrar funcionalidades que falten por plasmar en los diagramas. No comenzar la fase de desarrollo hasta que estos diagramas hayan sido correctamente revisados.
Plan de contingencia	Rediseñar y reimplementar lo antes posible los diagramas necesarios.

Tabla 35: Riesgo de bloqueo en el desarrollo

Riesgo de bloqueo en el desarrollo	
Descripción	Es posible que durante la fase de desarrollo surjan complicaciones debido a la dificultad inherente de implementar algunas de las tareas, produciéndose bloqueos en los que baja la productividad.
Probabilidad	Muy probable.
Impacto	Leve.
Plan de prevención	Estudiar a fondo las documentaciones de las herramientas necesarias, así como planear de antemano el trabajo que se va a realizar y cómo se va a realizar.
Plan de contingencia	En caso de ser posible, cambiar a otra tarea y, tras terminarla, retomar la actual. En caso de no ser posible, tomarse un pequeño descanso para retomar el problema con la mente clara un tiempo más tarde.

Tabla 36: Riesgo de cambios en las especificaciones

Riesgo de cambios en las especificaciones	
Descripción	Si fuese necesario redefinir las especificaciones del proyecto, es posible que parte del trabajo realizado quede inservible o necesite modificaciones, generando retrasos en la planificación.
Probabilidad	Poco probable.
Impacto	Alto.
Plan de prevención	No comenzar la fase de desarrollo hasta que las especificaciones hayan sido correctamente revisadas. Realizar una implementación lo más modular y encapsulada posible para, en caso de darse un cambio en las especificaciones, no sea necesario rehacer todo el código.
Plan de contingencia	Rediseñar y reimplementar lo antes posible aquellas partes del proyecto que se hayan visto modificadas.

2.5.3. Riesgos externos

Tabla 37: Riesgo de enfermedad o accidente

Riesgo de enfermedad o accidente	
Descripción	Abarca la posibilidad de que el desarrollador sufriera un accidente o una enfermedad que le impidiese realizar las tareas necesarias, pudiendo retrasar el proyecto.
Probabilidad	Improbable.
Impacto	Alto.
Plan de prevención	No es prevenible.
Plan de contingencia	Modificar la planificación temporal para reubicar el trabajo no realizado.

Tabla 38: Riesgo de productividad baja

Riesgo de productividad baja	
Descripción	Debido a trabajo o exámenes se puede disponer de menos tiempo del planteado para el desarrollo del proyecto, retrasándose éste como consecuencia.
Probabilidad	Muy probable.
Impacto	Alto.
Plan de prevención	Determinar una planificación temporal con una carga de trabajo diaria no demasiado alta, disminuyendo así la posibilidad de saturar el tiempo disponible.
Plan de contingencia	Modificar la planificación temporal para reubicar el trabajo no realizado.

2.5.4. Aplicación de los planes de prevención

Después de haber definido los riesgos es necesario aplicar los planes de prevención. En este proyecto hay dos planes de prevención compartidos entre diferentes riesgos a los que se dará una especial importancia: la realización de copias de seguridad y el desarrollo de un diseño suficientemente robusto.

En cuanto a las copias de seguridad, se plantea realizarlas de dos formas diferentes: por un lado, el proyecto será sincronizado de forma diaria mediante el control de versiones git en un repositorio en GitLab; por otro lado, y para mayor seguridad, cada dos semanas se realizará una copia de seguridad completa del proyecto en un disco duro externo.

Para cumplir con el desarrollo de un desarrollo de un diseño robusto, será necesario asegurarse de que se ha dedicado el suficiente tiempo a toda tarea que de lugar a un diagrama que se utilizará para el desarrollo software, así como a revisarla de forma exhaustiva.

2.6. Evaluación económica

Si bien no es el caso de este proyecto, pues se desarrollará como parte de un expediente académico, cuando se desarrolla un juego se espera cierto beneficio al sacarlo al mercado. Este beneficio puede surgir de diferentes formas de negocio. Sin embargo, antes de proponer y comparar diferentes opciones para recuperar la inversión, hemos de calcular el coste total del proyecto.

En un primer momento se dividirán los gastos en categorías y se analizarán detalladamente una por una. Tras calcular la inversión total, se propondrán diferentes modelos de negocio con los que se pretenderán obtener ganancias en el supuesto de que en un futuro Brawlchemy saliera al mercado.

2.6.1. Gastos en mano de obra

El trabajo realizado se dividirá en dos categorías principales: trabajo de programador y redactor; y trabajo de analista y diseñador. Según los datos del convenio de las TIC que figura en el BOE [1], el salario mínimo anual de un analista y diseñador software asciende a 25189.02€ (equivalentes a unos 14€ por hora), y el de un programador y redactor es de 17715.65€ (equivalentes a unos 8.85€ por hora).

En base a estos datos, y considerando que corresponden con la cota mínima de un salario en un trabajo de estas características, se harán los cálculos utilizando unas cantidades ligeramente mayores (1.5 veces mayor), simulando el primer salario al incorporarse a una empresa como recién graduado: 20€ por hora para el trabajo de analista y 12.50€ por hora para el trabajo de programador.

Definido el gasto por hora de trabajo, y teniendo en cuenta qué solamente las tareas pertenecientes a los módulos de «Captura de requisitos» y «Análisis y diseño» se consideran trabajo de un analista, obtenemos el total en base a la fórmula:

$$20€ \cdot (19 + 37) + 12{,}50€ \cdot (90 + 26 + 43 + 100) = 4357{,}50€$$

2.6.2. Gastos software

En cuanto al sistema operativo, el proyecto será realizado desde Ubuntu, una distribución de Linux totalmente gratuita. Además, todas las herramientas software elegidas para la realización de este proyecto son herramientas también gratuitas, siendo Aseprite la única excepción. Sin embargo, los desarrolladores de Aseprite disponen de código fuente abierto que cualquier usuario puede compilar y utilizar de forma gratuita. De la misma forma, entre los motores gráficos propuestos todos disponen de versiones gratuitas. Por tanto, este proyecto se ha realizado sin ningún gasto software asociado.

2.6.3. Gastos hardware

En la tabla 39 se pueden observar los gastos hardware de forma desglosada

Tabla 39: Gastos hardware del proyecto

Aparato	**Precio**	**Vida útil**	**Amortización en 3 meses**
Ordenador portátil	1100€	5 años	55€
Monitor secundario PC	150€	10 años	3.75€
Total			**58.75€**

La amortización representa el gasto en el tiempo de un bien con un valor duradero en el tiempo, como el hardware. Ésta ha sido calculada mediante la siguiente fórmula (vida útil expresada en meses):

$$Amortización\ mensual = \frac{Precio}{Vida\ útil} \tag{1}$$

Sabiendo cuál es la amortización mensual, la amortización durante el proyecto es equivalente a la multiplicación de ésta por la duración del proyecto. En este caso, la duración de 3 meses para el proyecto está extraída de la tabla 26.

2.6.4. Gastos totales

Teniendo en cuenta todos los gastos anteriores como conjunto podremos calcular el gasto total del proyecto. En la tabla 40 podemos observar los gastos parciales y el total, el cual asciende a los 4416,25€.

Tabla 40: Gastos totales del proyecto

Concepto	**Gasto**
Gastos en mano de obra	4357,50€
Gastos software	0€
Gastos hardware	58.75€
Total	**4416.25€**

2.6.5. Modelo de negocio

Una vez calculado el gasto total, podemos plantear diferentes alternativas de modelo de negocio. En el mercado de los videojuegos se contemplan tres modelos principales: la venta completa, la venta temporal y la publicación gratuita con compras dentro del juego.

Por un lado, existe la opción de venta completa del juego. Por un precio determinado, el usuario adquiere el derecho a la utilización de ese software, generalmente con todas sus funcionalidades. Si bien pudiera parecer la opción más inteligente , hay que tener en cuenta ciertos factores. En primer lugar se debería llevar a cabo una encuesta de mercado para calcular de forma aproximada las ventas que el juego podría tener. Ya con una cifra en manos, podríamos calcular de forma sencilla un precio mínimo a partir del cual obtendríamos beneficios mediante la fórmula $Precio = Coste/Ventas\ totales$. Además, también debemos tener en cuenta el abanico de precios en el que oscilan los videojuegos similares, puesto que un precio demasiado alto podría reducir las ventas y no ser razonable de cara a los precios de mercado establecidos y que los consumidores pueden considerar.

Por otro lado existe la venta temporal, en la que el usuario paga una cantidad de dinero por la utilización del juego durante un tiempo preestablecido. En cierta medida la preparación necesaria para este modelo es similar a la de la venta completa, aunque es más complejo determinar las ventas de esta forma. Este modelo está especialmente indicados a juegos diseñados para ser «adictivos», pues se fomentará a los usuarios a seguir pagando por el servicio y, en situaciones óptimas, se obtendrán mayores beneficios que con una venta completa.

Por último está el modelo de juego gratuito con compras internas. Normalmente se ofrece la posibilidad de comprar añadidos o modificaciones para el juego a través de una moneda virtual del juego, la cual se puede obtener a cambio de pequeñas cantidades de dinero real, comúnmente llamadas microtransacciones. Estas modificaciones suelen ser cambios o mejoras

estéticas y en ningún momento afectan a la jugabilidad más allá de modificar la apariencia. Esta opción es muy popular entre muchos tipos de juego, entre ellos los juegos del género MOBA (véase sección Bibliografíasec:antecedentes), puesto que es especialmente efectiva para juegos sin un final definido y que pueden ser jugados repetidas veces, ya que el modificar pequeñas partes del juego disminuye la sensación de repetición.

Además, existe la opción de utilizar varias de las estrategias de mercado a la vez, combinándolas en una gran variedad de formas. Por ejemplo, ciertos juegos de pago disponen a su vez de modificaciones estéticas adicionales adquiribles a través del propio juego, siendo una mezcla del primer y tercer modelo; o existiendo también juegos que deben ser comprados y que además ofrecen los conocidos *Season-Pass* (pases de temporada), contenido jugable adicional que se puede disfrutar durante un periodo de tiempo, tras el cual se elimina del juego con la introducción de un nuevo *Season-Pass*, siendo así una mezcla de los dos primeros modelos.

Vistas estas alternativas, se considera que la más acertada para el juego que se pretende crear es usar el modelo de juego gratuito con compras internas. Hoy en día este modelo está fuertemente establecido como la principal forma de ingresos en el sector [2][3]. Además, de cara al usuario medio, este se plantea como el método menos «intrusivo» y el que más gente atrae, pues se puede probar el juego de forma gratuita y, si este logra captar su atención, es muy probable que se inviertan pequeñas cantidades de dinero en él.

3. Antecedentes

En cualquier proyecto de desarrollo software es conveniente comprobar si algún proyecto anterior ya ha cubierto los objetivos que el nuestro pretende cumplir. Si bien esto puede ser un problema en un proyecto software típico, pues no queremos usar demasiados recursos en algo que ya está hecho por otro desarrollador, cuando hablamos sobre videojuegos, es difícil desarrollar algo que nunca nadie haya hecho antes, o al menos algo muy similar. En una industria tan explotada como es la del videojuego, la innovación generalmente se reserva a pequeñas o medianas modificaciones a géneros arquetípicos.

Es aquí donde se pretende ubicar este proyecto. Habiendo escogido el género MOBA, se pretende añadir funciones o mecánicas de las que otros juegos similares no dispongan. Para ello, se listarán a continuación los principales juegos del género y se hará una comparativa entre ellos para determinar cuáles son las principales diferencias y en qué aspectos hay posibilidad de innovación.

3.1. League of Legends

League of Legends es el principal representante en la actualidad del género MOBA, siendo con mucha diferencia el que tiene más jugadores diarios [4]. Lanzado en 2009, es uno de los primeros grandes MOBA y el que más popularidad ha dado al género de cara al público. Cada equipo consta de 5 jugadores, con una vista *top-down* y un entorno tridimensional. Se dispone de una gran variedad de personajes (143 a fecha de Marzo de 2019).

League of Legends es completamente gratuito y principalmente genera sus beneficios mediante la compra de skins, cambios gráficos en los personajes jugables que no alteran de ninguna forma la funcionalidad de los mismos, es decir, son modificaciones meramente estéticos. [5]

3.2. DOTA 2

DOTA 2 es la evolución del primer MOBA como lo conocemos hoy en día, *Defense of the Ancients*. Salió al mercado en 2010 y hoy en día es el segundo MOBA más jugado, sólo superado por League of Legends [4]. De forma similar a League of Legends, los equipos cuentan con 5 personas, pudiendo elegir cada una entre 120 personajes diferentes. La vista es *top-down* y el entorno es tridimensional.

Como es habitual en el género, es un juego gratuito que genera ingresos mediante la venta de objetos cosméticos para los personajes.

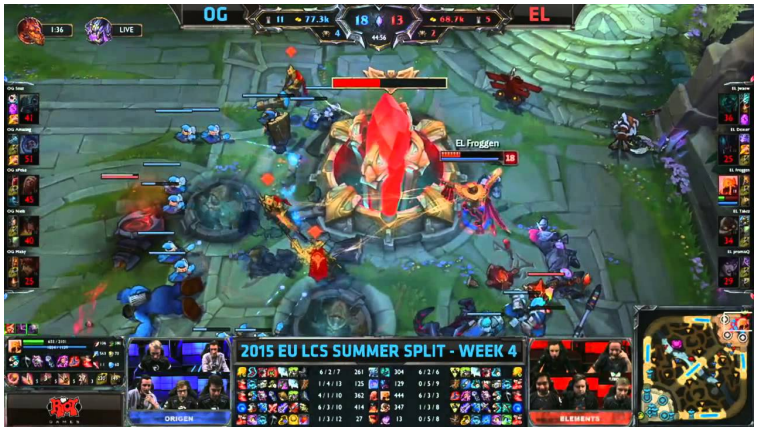


Figura 4: Partida profesional de League of Legends [6]

Figura 5: Partida de Dota 2 [7]

3.3. Heroes of the Storm

Heroes of the Storm es uno de los últimos grandes MOBA, pues fue publicado en 2015 por parte de Blizzard. Es muy similar a los anteriores en cuanto a su vista *top-down*, su entorno tridimensional y sus equipos de 5 jugadores. Su principal atractivo de cara al público radica en que los personajes a utilizar son pertenecientes a otras famosas franquicias de Blizzard como Warcraft o Starcraft, por lo que el juego es una mezcla de personajes de diferentes «universos». Cabe destacar que, a diferencia de la mayoría de MOBA, no existe un sistema de equipamiento dentro de la partida, es decir, poder adquirir objetos durante cada partida que harán a tu personaje progresivamente más fuerte. En el resto de MOBA este sistema hace que quien va ganando tenga más posibilidades de aumentar su poder y ganar por una diferencia mayor, lo que se conoce como efecto bola de nieve. La ausencia de dicho sistema en Heroes of the Storm hace que las partidas sean más equilibradas a lo largo del tiempo, siendo más complicado para un equipo conseguir una ventaja muy grande.

Al igual que los anteriores, Heroes of the Storm es gratuito con la posibilidad de comprar mejoras estéticas dentro del juego. [8]

Figura 6: Partida de Heroes of the Storm [9]

3.4. Awesomenauts

Si bien Awesomenauts no es uno de los MOBA más conocidos ni jugados, destaca en cuanto a su presentación, pues se trata de uno de los pocos ejemplos de MOBA en dos dimensiones y de plataformas, además de tener los equipos 3 jugadores cada uno. Esto hace que la experiencia sea muy diferente a otro tipo de MOBA, y lo enfoca mucho más a la acción y la velocidad en vez de a la pura estrategia.

Awesomenauts es completamente gratuito, pudiendo comprar skins para los personajes mediante dinero real. [10]

Figura 7: Partida de Awesomenauts [11]

3.5. Smite

Smite conserva todas las mecánicas clásicas del MOBA, pero se diferencia del resto en cuanto a que es un juego en tercera persona, es decir, controlamos a nuestro personaje desde «su espalda» en un entorno completamente tridimensional. Esto modifica en gran medida la forma en la que se juega, pues no dispones de una visión general del mapa en prácticamente todo momento, sino que te limitas a lo que tu personaje está observando.

Como ya es común, se puede jugar Smite de forma gratuita, disponiendo de compras de mejoras cosméticas dentro del juego. [12]

Figura 8: Partida de Smite [13]

3.6. Resumen de los antecedentes

Para poder comparar de forma sencilla los diferentes MOBA discutidos, se han definido ciertos criterios:

- **Nº jugadores:** Número de jugadores totales que participan en cada partida.
- **Motor gráfico:** Hace referencia al tipo de renderizado del juego, pudiendo ser este 2D o 3D.
- **Perspectiva:** Lugar donde se ubica la cámara en base a la posición del jugador.
- **Experiencia y nivel:** Existencia de progresión a lo largo de la partida, haciendo que el personaje se haga más fuerte de forma pasiva en función de las veces que mate a los enemigos o alcance diferentes objetivos.
- **Mejoras in-game:** Posibilidad de mejorar el personaje mediante la adquisición de objetos. Similar a la Experiencia y nivel, pero la mejora se realiza de una forma activa por parte del jugador y es opcional.

En la tabla 41 se puede ver una comparativa entre los MOBA discutidos anteriormente y sus principales diferencias:

Tabla 41: Comparativa diferentes MOBA

Juego	Nº jugadores	Motor gráfico	Perspectiva	Experiencia y nivel	Mejoras in-game
League of Legends	5	3D	Top-Down	Sí	Sí
Dota 2	5	3D	Top-Down	Sí	Sí
Heroes of the Storm	5	3D	Top-Down	Sí	No
Awesomenauts	3	2D	Lateral	No	Sí
Smite	5	3D	Tercera persona	Sí	Sí

Como se puede observar, el género MOBA está muy estandarizado y es raro que se creen juegos que innoven en él. Sin embargo, se aprecia que es poco común encontrar MOBA en 2D, con un enfoque más orientado hacia el género de plataformas. De igual forma, la mayoría de juegos existentes apuestan por un sistema que incluya tanto experiencia y niveles como la posibilidad de adquirir mejoras dentro de las partidas. Esto hace que en general sea difícil iniciarse en el género, pues la complejidad que crean estos sistemas puede llegar a ser abrumadora para usuarios menos avanzados.

Es aquí donde entra en juego este proyecto, pues encaja perfectamente en ese pequeño nicho que está tan poco explorado, pues pretende ser un MOBA en 2D simplificado, siendo así fácil de entender y disfrutar para cualquier usuario.

3.7. Elección del motor

Entre los diferentes motores gráficos listados en el apartado Herramientas, y teniendo en cuenta los aspectos discutidos en esta sección, es necesario decantarse por una de las opciones para el desarrollo de Brawlchemy.

En primer lugar, se descartarán las dos opciones menos indicadas: Cocos2d y GameMaker Studio 2. En primer lugar, si bien Cocos2d está especialmente diseñado para juegos 2D como Brawlchemy, no ofrece suficientes funcionalidades para el desarrollo de un juego tan complejo como un MOBA en el aspecto del modo multijugador. Por otro lado, GameMaker Studio 2 sí que dispone de librerías para implementar esta funcionalidad, pero no están completamente implementadas en el propio motor y es necesario un gran trabajo extra para conseguir la funcionalidad. Además el motor es un tanto limitado en cuanto al desarrollo, pues su modo de edición *drag and drop* no se puede sustituir completamente por desarrollo a base de scripts.

Las opciones restantes son Godot y Unity, dos motores muy similares en cuanto a estructura y desarrollo. Para tomar una decisión se han comparado ambos en base a las licencias que ofrecen, los lenguajes de programación que utilizan, su optimización para juegos 2D, su integración con sistemas de control de versiones y la documentación e información disponible acerca de ellos:

- La licencia de Godot es completamente gratuita, por lo que no es necesario prescindir de ciertas funcionalidades como en Unity.
- Unity es un motor mucho más popular que Godot, y como tal dispone de una mayor cantidad de información online acerca de su uso, así como tutoriales o ejemplos de algunas funcionalidades. Si bien Godot dispone de una muy buena documentación, es más difícil encontrar información sobre su funcionamiento fuera de dicha documentación al ser un motor gráfico relativamente nuevo
- La optimización en Godot es mejor para juegos 2D, pues mientras que Unity ejecuta su modo 2D como una extensión del 3D, Godot dispone de modos completamente independientes. Esta diferencia puede ser crucial cuando el sistema se encuentre bajo un carga de trabajo alta.
- El lenguaje de scripting GDScript, similar a Python, es mucho más simple, legible, y eficiente de cara al desarrollo de grandes piezas de código en comparación a C#.
- Godot compila sus ficheros de forma que sean compatibles con sistemas de control de versiones como Git, pues estos pueden ser leídos como texto plano. Esto no es así con Unity, el cual genera ficheros binarios que pueden ser incómodos a la hora de mantener un repositorio lo más limpio y ordenado posible.

Entre ambos se ha decidido utilizar Godot, puesto que aunque sea más complicado encontrar ejemplos de su uso, el resto de aspectos positivos comentados anteriormente superan con creces a los de Unity. Además, se considera importante fortalecer la capacidad de leer y entender documentación oficial de diferentes herramientas y trabajar en función a estas, por lo que un punto que en principio puede suponer una desventaja puede ser utilizado para mejorar las capacidades personales en ese aspecto.

4. Captura de requisitos

En esta sección se trata la captura de requisitos de Brawlchemy. Esto engloba la definición de los requisitos funcionales mínimos que se deben cumplir al finalizar el proyecto, de los diferentes roles que pueden tomar los usuarios al utilizarlo y las acciones que puede realizar cada rol, y de la información que se almacenará en el sistema.

Esta sección es de especial importancia, pues es en una captura de requisitos sólida donde se sustenta una buena planificación y desarrollo de un proyecto.

4.1. Requisitos no funcionales de Brawlchemy

A continuación se listarán los requisitos no funcionales que Brawlchemy debe satisfacer.

- Cada jugador controlará a un personaje propio.
- Los personajes inicialmente aparecerán en distintas ubicaciones del mapa (*spawn*) en función del equipo en que participen.
- Los personajes disponen de una salud máxima. Esta salud se puede perder por causa de ataques de jugadores enemigos, y se recuperará progresivamente después de pasar un tiempo sin recibir daño. Cuando la salud de un personaje se reduzca a 0, este morirá. Tras ello, y al cabo de un tiempo, el personaje aparecerá en su *spawn* correspondiente.
- Cada personaje dispone también de un recurso secundario: energía. La energía actúa como recurso limitador de los jugadores, pues se utilizan ciertas cantidades de ésta para llevar a cabo algunas acciones del personaje, como correr, saltar o atacar. De un modo similar a la salud, la energía comienza a recuperarse progresivamente tras no ser utilizada durante un tiempo.
- Cada tipo de personaje dispondrá de diferentes valores de salud y energía, así como habilidades con las que podrá atacar únicamente a miembros o estructuras del equipo contrario. En este proyecto se implementarán 4 tipos de personaje: un mago, un arquero, un guerrero y un boxeador.
- Cada equipo dispondrá de varias torres, estructuras estáticas que servirán como defensa de la base de dicho equipo. Estas torres atacarán a miembros del equipo contrario siempre que un miembro del equipo aliado se encuentre cerca de ellas, y podrán y deberán ser destruidas para poder llegar a la base del equipo al que defienden.
- Cada equipo dispondrá de una estructura principal en su base, el nexo. La partida finalizará cuando uno de los equipos destruya el nexo enemigo.

4.2. Jerarquía de actores

El juego dispone únicamente de dos actores, representados en la figura 9:

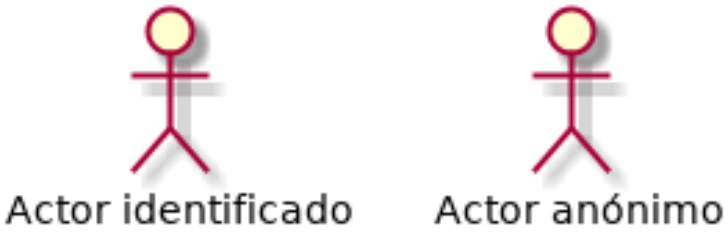


Figura 9: Jerarquía de actores

- **Actor anónimo:** Comprende a los usuarios que accedan al juego y aún no se hayan identificado.
- **Actor identificado:** Representa a los usuarios que han iniciado sesión en el juego con sus datos de autentificación.

4.3. Casos de uso

Habiendo definido ya los requisitos del sistema y los diferentes actores que van a interactuar con él, un diagrama de casos de uso representa gráficamente las interacciones entre los actores y el sistema para llevar a cabo las funcionalidades de este último. En la figura 10, se muestran los principales casos de uso de Brawlchemy. En el anexo A se muestran los casos de uso extendidos, en donde se analiza cada uno de ellos a fondo.

A continuación se explicará la funcionalidad a la que hace referencia cada caso de uso:

- **Registrarse:** Permite al usuario crear una nueva cuenta en el sistema.
- **Iniciar sesión:** Permite al usuario utilizar su cuenta para iniciar sesión en el sistema.
- **Jugar:** Permite al usuario acceder al menú en el que se gestionan las partidas disponibles.
- **Crear partida:** Permite al usuario crear una nueva partida a la que unirse.
- **Elegir partida:** Permite al usuario unirse una partida disponible.
- **Elegir personaje:** Permite al usuario elegir el personaje con el que jugará la partida.

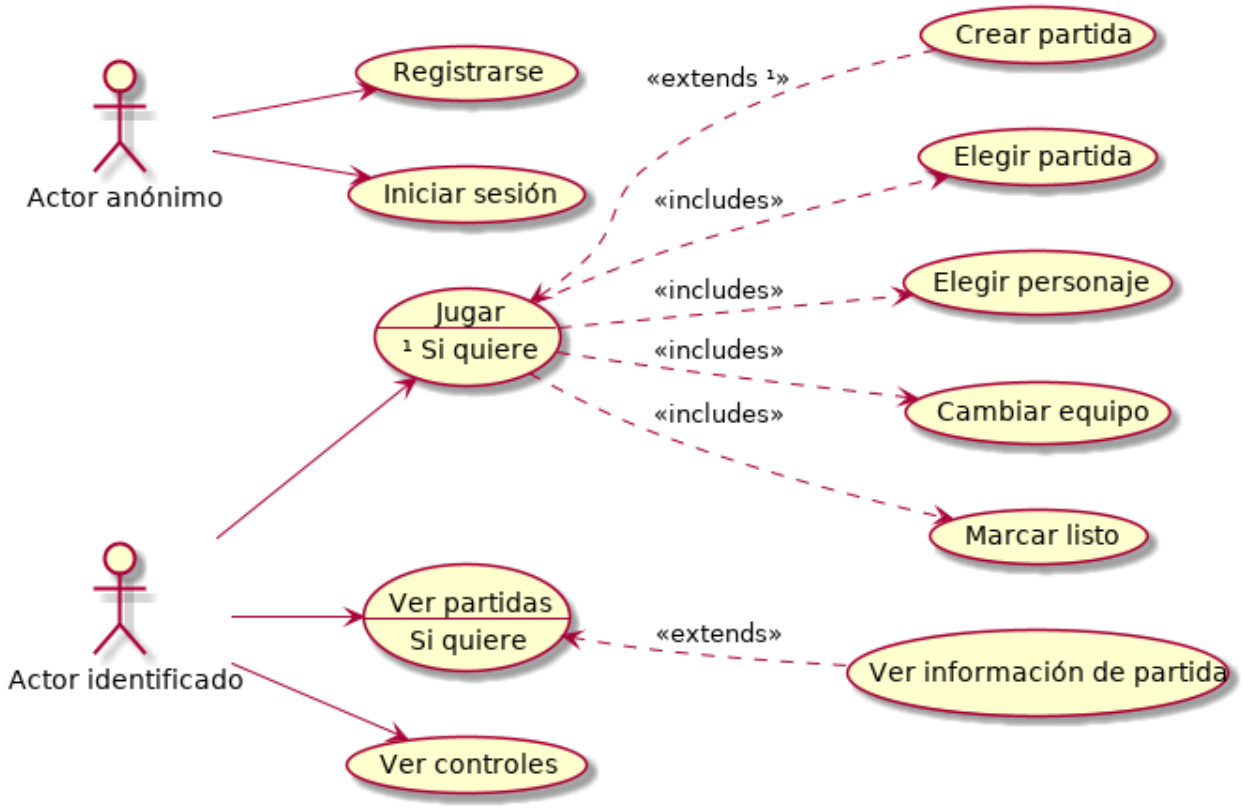


Figura 10: Casos de uso de Brawlchemy

- **Cambiar equipo:** Permite al usuario cambiar el equipo con el que jugará la partida.
- **Marcar listo:** Permite al usuario comunicar al juego que está preparado para que la partida comience.
- **Ver partidas:** Permite al usuario visualizar una lista de las partidas en las que ha participado.
- **Ver información de partida:** Permite al usuario ver información sobre una partida en la que ha participado.
- **Ver controles:** Permite al usuario ver los botones y teclas con los que se controla el juego.

4.4. Modelo de dominio

Mediante el modelo de dominio se trata de representar de forma conceptual la estructura de los registros y eventos del sistema en forma de objetos o entidades con sus correspondientes atributos, además de mostrar también todas las relaciones existentes entre estos. El modelo surge de las especificaciones y necesidades del proyecto, y con éste se podrá comprender de forma general el sistema para más adelante desarrollar un análisis conciso y acertado.

En el caso de este proyecto, mediante el modelo de dominio se representará a todos aquellos usuarios que jueguen a Brawlchemy, así como las partidas a las que estos se unen y en las que participan.

En la figura 11 se puede ver el diagrama del modelo de dominio del servidor de Brawlchemy.

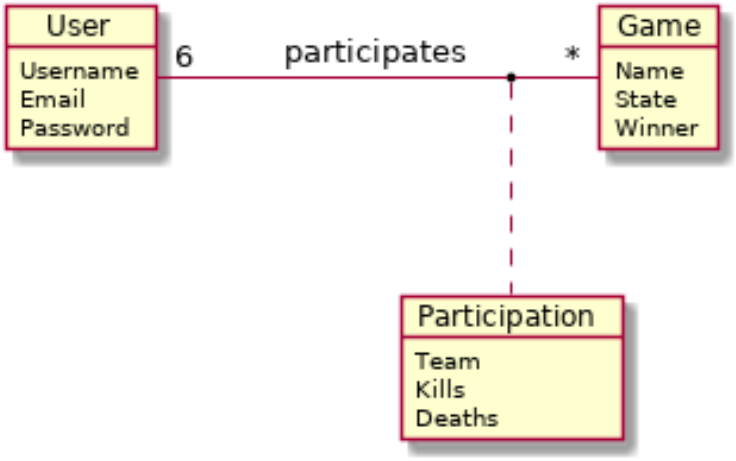


Figura 11: Modelo de domino de Brawlchemy

Como se puede observar, existen dos entidades:

- **Usuario:** En la entidad Usuario solamente se almacenan los datos necesarios del usuario para su registro y autentificación en Brawlchemy: un nombre de usuario (atributo «Username»), un correo electrónico (atributo «Email») y la contraseña que utilizarán para iniciar sesión una vez registrados (atributo «Password»).
- **Partida:** En la entidad Partida se almacenan la información de las partidas que tienen lugar en el servidor: el nombre de ésta (atributo «Name»), el estado en que se encuentra (atributo «State»): en preparación, siendo jugada o terminada; y, en caso de que la partida haya terminado, el equipo ganador (atributo «Winner»).

Por otro lado, se da una relación entre entidades, teniendo esta asociación unos atributos:

- **Participación (Usuario - Partida):** Esta relación representa la participación de un usuario en una partida. Como se puede ver, en una partida tienen que participar 6 personas, y cada usuario puede participar en todas las partidas que desee. Además, se almacenará el equipo en el que cada jugador participa (atributo «Team»), así como algunas estadísticas de cada usuario en dicha partida: las veces que ha matado a jugadores enemigos (atributo «Kills») y cuántas veces lo han matado estos a él (atributo «Deaths»).

5. Análisis y diseño

En esta sección se trata el análisis y diseño de Brawlchemy. Se utilizará la información obtenida en la captura de requisitos para definir de forma concisa la estructura del proyecto para posteriormente utilizarla como base del desarrollo de éste.

5.1. Diagrama relacional de la base de datos

En el capítulo anterior (sección 4.4) se ha descrito el modelo de dominio de Brawlchemy. En base a este modelo se puede desarrollar el diagrama relacional de la base de datos que se usará en el servidor de Brawlchemy. Éste se puede observar en la figura 12:

Figura 12: Diagrama relacional de la base de datos de Brawlchemy

Como se puede ver, el diagrama relacional de la base de datos sigue una estructura muy similar al modelo de dominio. A continuación se describirán las tablas resultantes de la transformación y sus atributos:

- **Usuarios (user):** Almacena la información de los usuarios registrados en el sistema. Aparte de los datos introducidos por el propio usuario para el registro (nombre de usuario, correo electrónico y contraseña) se utiliza como clave un campo ID que identifica a cada uno de forma única, siendo éste asignado de forma automática al crear un nuevo usuario.
- **Partidas (game):** Almacena la información de las partidas del sistema. Éstas tienen un nombre, un estado, un puerto y un ganador. El nombre es asignado por el usuario que crea la nueva partida. El estado representa mediante un número en cuál de los diferentes estados se encuentra la partida (0: en espera, 1: en proceso, 2: terminadas, -1: erróneas/otros). El campo puerto indica a través de que puerto deben conectarse los jugadores que quieran unirse a la partida o, en caso de haber terminado ya, el puerto en que se jugó. El campo del ganador representa al equipo que se hizo con la victoria (0: equipo azul, 1: equipo rojo, -1: aún no hay ganador).

- **Participaciones (participation):** Almacena la información de la participación de un jugador dado en una partida concreta. Los campos «user_id» y «game_id» hacen referencia al identificador del jugador y al de la partida en la que se participa respectivamente. El campo equipo representa el equipo en el que participa el jugador (igual que en el campo de la tabla partida: 0: equipo azul, 1: equipo rojo). El campo kills almacena el número de veces que el jugador ha matado a un enemigo, y el campo deaths el número de veces que éste ha muerto.

5.2. Estructura del juego

En esta sección se describirá la estructura que compone Brawlchemy. Para comprender ésta completamente es recomendable leer el anexo C «Introducción a Godot» , en el que se explican las características del desarrollo en Godot y algunas de sus particularidades. El desarrollo de los modelos que definirán la estructura del proyecto se ha divido en dos partes: los modelos de los menús y los del propio juego.

5.2.1. Menús

Godot dispone de una serie de nodos cuya función está orientada específicamente a la construcción de menús. Muchos de estos nodos están presentes en otros lenguajes de programación, como botones, campos de texto o etiquetas de texto. Además, Godot proporciona al desarrollador los «contenedores», un tipo de nodo que sirve para organizar sus nodos hijos en función de una serie de disposiciones diferentes (horizontalmente, verticalmente, en forma de lista, etc). En todos los menús a que se construirán en Brawlchemy se utilizarán principalmente estos nodos, para dotar así a las interfaces del aspecto deseado., siendo éste el que se definió anteriormente en los casos de uso extendidos (véase anexo A).

Los menús del juego dispondrán de dos funcionalidades principales: por un lado permitirán, mediante botones, moverse a través de las escenas de las diferentes interfaces (registro, login, menú principal, etc); por otro lado, podrán realizar peticiones al servidor recibir información que se mostrará al usuario o para enviar información que el usuario ha introducido.

Para realizar la primera función es necesario sustituir la escena que se está mostrando al usuario, definida en el árbol general, por lo que el desarrollo de ello se puede realizar de forma simple. Para enviar información a través de peticiones HTTP se dispone de un nodo cuya función es el envío de éstas, llamado «HTTPRequest». Este nodo permite crear y enviar la petición, y dispone de una serie de señales que se emiten cuando esta petición termina. Mediante la creación en tiempo de ejecución de estos nodos es posible realizar diferentes peticiones y reaccionar a los resultados de éstas.

La forma de realizar estas peticiones, así como su estructura y los datos que devuelven se explica más adelante (apartado 5.3).

5.2.2. Juego

Además de los menús, es necesario definir la estructura de la propia partida de Brawlchemy, siendo esta mucho más compleja que los menús al no disponer de nodos específicos que implementen de serie las funcionalidades necesarias. Tal y como se ha explicado en el anexo C, el desarrollo en Godot y, por tanto, la estructura que adquieren los proyectos es muy diferente a otros sistemas y lenguajes de programación. Si bien es virtualmente posible crear una especie de diagrama de clases (o de escenas y nodos, en este caso) completamente preciso en tanto a que contiene todos los nodos (junto a sus respectivos métodos) que forman una escena, el hecho de que éstas estén compuestas por una gran cantidad de nodos y escenas en cascada hacen que sea imposible en términos de complejidad, legibilidad y, sobre todo, utilidad.

Por ello, para este apartado se ha realizado una simplificación de este diagrama, pues en él se mostrarán sólo aquellas escenas y/o nodos principales. Para poder decidir qué nodos y/o métodos no serán mostrados en el diagrama, se ha definido una serie de reglas para excluirlos:

- No aparecerán métodos básicos de los nodos, salvo que su utilización fuera especialmente relevante. En esta categoría se incluyen principalmente dos métodos que son usados en todos los nodos: el método «_ready», llamado cuando el nodo es creado o renderizado en la escena por primera vez, siendo donde generalmente se ubican los métodos de inicialización; y el método «_process», siendo éste el que es llamado en cada frame de ejecución por el motor y donde se incluye toda la funcionalidad cíclica y continua, como el movimiento de los personajes o la interacción de estos con otros nodos.

- No se mostrarán nodos cuya existencia pueda ser deducida por el propio tipo de escena. Por ejemplo, cualquier entidad que aparezca en el juego y que el usuario deba ver en pantalla (como el personaje de un jugador) tendrá un sprite como nodo hijo, puesto que es el sprite el nodo que contiene la imagen que el usuario ve.

- No se incluirán nodos cuya existencia se pueda deducir por otros métodos o funcionalidades. Por ejemplo, si una escena dispone de un método para realizar comprobaciones en su entorno para detectar la existencia de otras escenas, se puede deducir que entre sus componentes se encuentra un área de detección. Del mismo modo, si un método se debe ejecutar en intervalos periódicos de tiempo, la existencia de un nodo temporizador es necesaria para el correcto funcionamiento del método.

Mediante esta serie de reglas simples es posible reducir el diagrama para que contenga solamente la información básica necesaria y pueda ser comprendido de forma sencilla. En la imagen 13 se muestra el diagrama que ha sido creado.

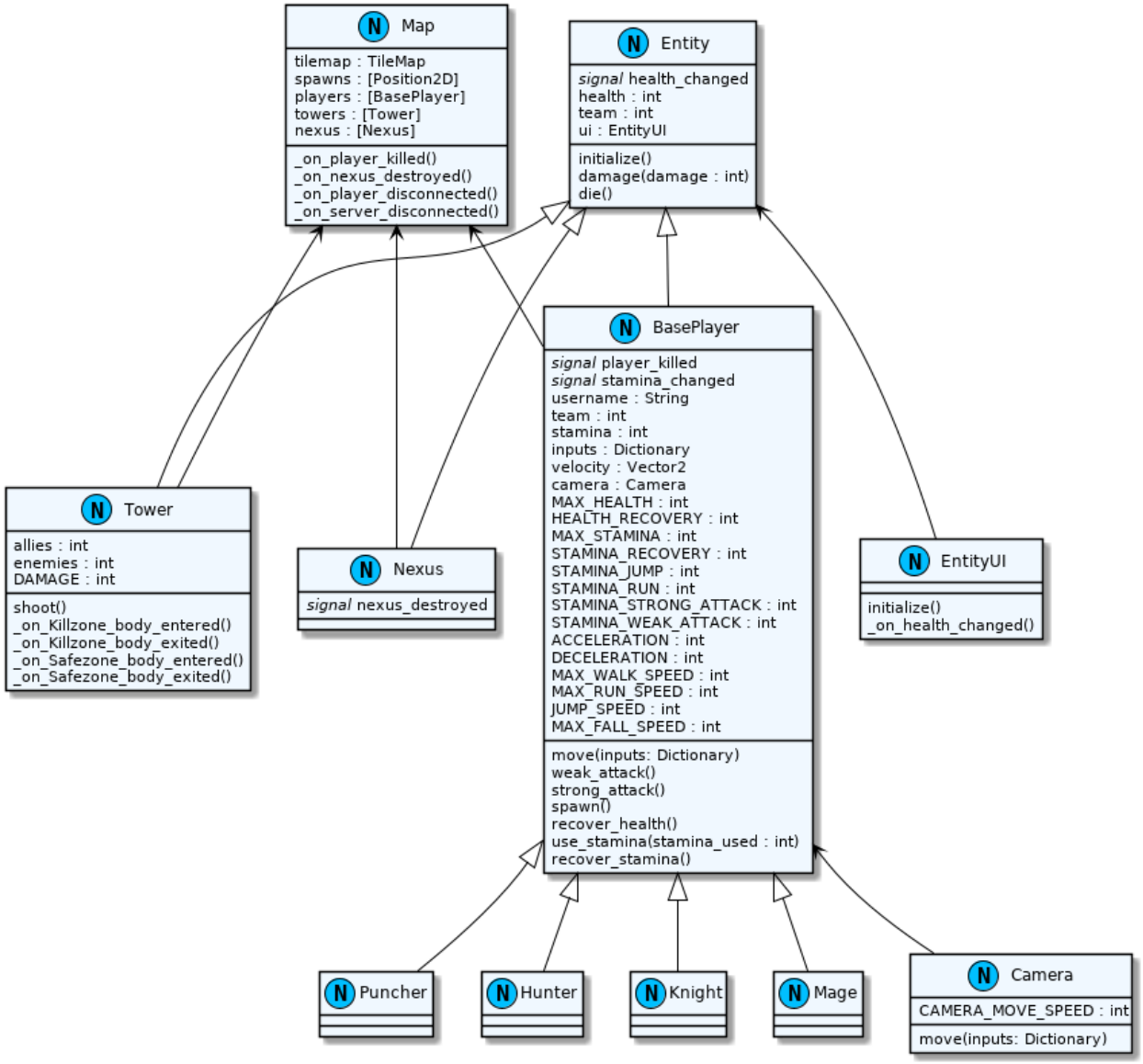


Figura 13: Diagrama de clases del juego

A continuación se explica la función de cada una de las escenas que se muestran en el diagrama:

- **Map:** El mapa es el nodo raíz de la partida. En él se ubican las distintas entidades que participan en ella (sean jugables o no) y el terreno. Además, es el mapa el que se encarga de la inicialización de los personajes, responde a los eventos que se dan entre ellos y gestiona el estado de la partida.
- **Entity:** Se llama entidades a todas las escenas que interactuarán entre ellas a lo largo

de la partida. Éstas disponen de una cantidad fija de vida, el equipo al que pertenecen, así como métodos para dañarlas y matarlas. También tienen un método desde el que se pueden inicializar algunos de sus valores, y servirá para que éstas puedan ser añadidas y gestionadas dinámicamente en tiempo de ejecución.

- **Tower:** Las torres son la principal estructura de defensa de los equipos. Éstas pueden detectar a los enemigos y a los aliados a su alrededor y atacarán a los enemigos siempre que un aliado se encuentre en las inmediaciones.

- **Nexus:** Los nexos son el objetivo final de cada equipo para ganar la partida. Estos sólo disponen de una señal que comunica al resto de nodos que han sido destruídos en caso de que un equipo lo logre, para así poder terminar la partida.

- **BasePlayer:** Esta escena será la base para la implementación de los diferentes tipos de personajes. Dispone de los datos del usuario que lo controla, de constantes que servirán para ejecutar los movimientos (stamina o «energía», velocidades máximas, aceleración y frenado, y velocidad de salto entre otros). Entre sus métodos principales se encuentra el que gestiona el movimiento en base a los inputs del jugador (los botones que éste está pulsando), los que ejecutan los ataques débiles y fuertes, o los que sirven para regenerar salud y energía.

- **Puncher, Hunter, Knight y Mage:** Estos son los diferentes personajes que podrán ser seleccionados para jugar a Brawlchemy. Todos ellos heredan sus nodos y métodos de BasePlayer, y los extienden con la implementación adicional necesaria para crear distintos tipos de personaje con diferentes habilidades.

- **EntityUI:** Esta escena es la que servirá para mostrar gráficamente la información de la entidad, como una barra de vida para representar su salud actual o una etiqueta para mostrar su nombre (en caso de tenerlo).

- **Camera:** Esta escena es la encargada de seleccionar lo que se muestra en la pantalla del usuario. Cada personaje tendrá una cámara como hijo, sirviendo ésta para que el jugador que lo controla pueda mantener su pantalla sobre dicho personaje. Además, este jugador podrá mover la cámara para mirar alrededor del propio personaje.

5.3. Estructura REST

Para la realización de las conexiones al servidor se ha considerado muy conveniente utilizar una arquitectura REST. En este apartado se definirá su estructura y funcionalidad, así como los motivos por los que se ha decidido utilizar este modelo.

En primer lugar, es necesario explicar en qué consiste este tipo de arquitectura. La Transferencia de Estado Representacional, o por su nombre más habitual REST (siglas en inglés de *REpresentational State Transfer*), es una arquitectura que define una serie de restricciones

para el desarrollo de servicios web. Esta arquitectura está completamente basada en el protocolo HTTP. Su principal ventaja frente a otros tipos de arquitecturas es que la información solicitada se encuentra en la propia petición HTTP de una forma simple para el usuario y los servicios que interactúan con los servidores que la implementan. Otra de las cualidades es que no se almacena ninguna información de los clientes en el servidor, por lo que en caso de necesitar permisos para realizar ciertas peticiones estos deberán ser provistos por el cliente en las propias peticiones.

Si bien el servidor web que va a utilizar esta aplicación no es demasiado grande y podría ser desarrollado de formas más sencillas sin utilizar estructuras internamente complejas como REST, de cara a las conexiones un esquema REST simplifica las tareas. Además, la arquitectura REST es fácilmente escalable, por lo que en caso de querer ampliar la aplicación en un futuro el trabajo sería mínimo. Por último, el hecho de que sea el cliente el que almacene la información necesaria simplificará que se mantenga una sesión iniciada mediante el envío de un token proporcionado por el propio servidor. Este token sólo será válido durante un tiempo preestablecido, tras el cual caduca. En caso de que el token del que dispone haya caducado, el usuario necesitará volver a iniciar sesión para recibir un nuevo token válido.

Por poner un ejemplo del funcionamiento de la arquitectura REST de una forma aplicada al servidor de Brawlchemy, supongamos que un usuario quiere ver la información de las partidas en las que ha participado. Suponiendo también que el servidor está alojado en «https://brawlchemy.info» y que el token de autentificación es «qwertyuiop». Esta información podría ser lograda mediante la siguiente petición:

```
Petition:
  GET https://brawlchemy.info/games
Headers:
  Authorization: qwertyuiop
```

En el ejemplo anterior se puede ver que en la propia dirección a la que se hace la petición está expuesta la información que se solicita. Si necesitásemos acceder a la información de una partida concreta con id «1234», la petición quedaría de la siguiente forma:

```
Petition:
  GET https://brawlchemy.info/games/1234
Headers:
  Authorization: qwertyuiop
```

Con estos dos ejemplos se puede ver de una forma más clara la flexibilidad que ofrece esta arquitectura en el desarrollo de un servicio web y la simplicidad de cara a los usuarios y desarrolladores que la utilicen.

En la figura 14 se muestra el modelo REST del servidor, con todas las peticiones posibles, indicando el tipo de petición (negrita) y los parámetros necesarios (en azul); y los códigos (negrita) y datos que éstas devuelven en caso de éxito (en verde) y en caso de error (en rojo). Los datos en formato JSON se encuentran representados entre llaves y corchetes (objetos o arrays respectivamente).

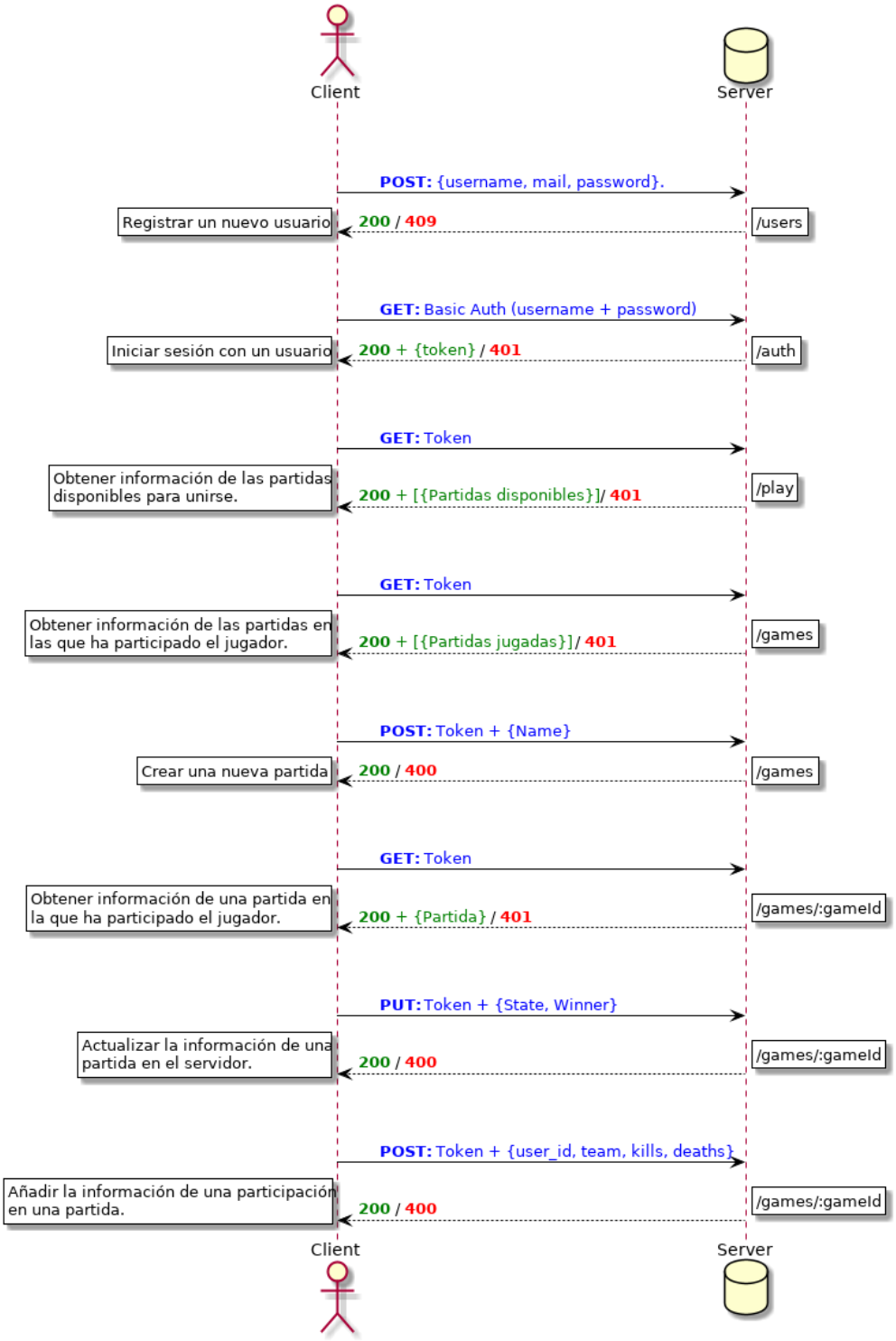


Figura 14: Modelo REST del servidor de Brawlchemy

6. Desarrollo

En esta sección se muestran los pasos seguidos para el desarrollo de cada uno de los prototipos del proyecto, así como las pruebas realizadas para cada uno de ellos.

Como se explicó anteriormente, el desarrollo consta de 5 prototipos diferentes:

- **Base de datos y servidor:** En él se creará el servidor que permitirá la conexión de los clientes al juego.
- **Menús, login y registro:** En él se implementarán los menús que permitirán registrarse, iniciar sesión y acceder a las diferentes funcionalidades de Brawlchemy.
- **Creación de personajes jugables:** En él se crearán la base de los personajes jugables, así como el mapa y otros elementos con los que interactúan estos.
- **Control del estado de la partida:** En él se llevará a cabo la implementación de un sistema que controle si una partida ha terminado y almacene su resultado en el servidor.
- **Diseño final de personajes y gráficos:** En este último prototipo se crearán los diferentes personajes y sus habilidades, y se añadirán los gráficos finales a aquellas escenas que lo necesiten.

6.1. Base de datos y servidor

Durante el desarrollo de este prototipo se creará e implementará el servidor en el que se alojará tanto el juego como la base de datos, así como las conexiones que se realizarán a esta última desde el juego por medio de conexiones HTTP a una API REST.

6.1.1. Desarrollo

En primer lugar, para poder disponer de un servidor se debe tener una maquina virtual. Para esto, desde la página web de Google Cloud Engine, se crea un nuevo proyecto al que se da el nombre «Brawlchemy».

Una vez creado el proyecto y encontrándonos dentro de él, se selecciona la pestaña *Compute Engine*, y se podrá acceder a las instancias de máquinas virtuales del proyecto. En este momento no hay ninguna, por lo que se crea una nueva. Cloud Engine da la opción de seleccionar diferentes sistemas operativos, potencia del procesador, memoria, etc. En este caso se ha escogido Ubuntu Server como sistema operativo, debido a que es el sistema en que se va a desarrollar el proyecto localmente. En el resto de opciones se ha seleccionado

la opción más conservadora, pues Google ofrece este servicio de forma gratuita durante un número determinado de horas al mes siempre y cuando se elija la máquina menos potente.

Tras esto disponemos de una máquina virtual a la que, utilizando las credenciales de Google, se podrá acceder mediante SSH y enviar archivos mediante SCP. A partir de este punto, para simplificar el desarrollo, los pasos a realizar sobre la máquina virtual serán primero realizados en el ordenador que se utiliza para desarrollar el proyecto, pudiendo así probar su funcionamiento de forma local de la misma forma que se podría probar en la máquina virtual. Una vez termine el desarrollo del servidor, todos los archivos necesarios serán copiados a la máquina virtual. Con esto se simplifica la tarea de crear el código necesario al tener editores de texto de forma local, y se ahorra tiempo al no tener que copiar continuamente lo que de va implementando.

El primer paso es instalar MySQL y node. Tras terminar la instalación, un asistente en la línea de comandos indica los pasos necesarios para configurar el usuario root de MySQL. Una vez acabe, se dispone de una configuración totalmente funcional de MySQL. En este momento se ha de crear la base de datos «brawlchemy», así como un usuario especial que sólo disponga de permisos sobre esta base de datos. Este usuario será el que se utilice desde el servidor web para acceder a la base de datos y realizar las diferentes operaciones, puesto que utilizar un usuario con permisos sobre todo el sistema, como lo es root, sería un riesgo de seguridad. Con la creación de este nuevo usuario, al que llamaremos «server», se minimizan los posibles efectos negativos en caso de que se diese una brecha de seguridad.

Una vez creada la base de datos y el usuario con permisos sobre ella, se han creado las tablas de la base de datos según se definieron al crear el diagrama relacional de la base de datos. Con esto, toda la configuración necesaria para MySQL está terminada.

Es el momento de preparar ser servicio web mediante NodeJS. Una vez instalado node, se crea una carpeta donde se ubicarán todos los archivos necesarios para el servidor. Mediante el comando «npm init» se lanza un instalador. Siguiendo los pasos que éste muestra por pantalla se puede dar nombre al proyecto, guardar la información del creador y otros datos secundarios, tras lo cual se crea la estructura básica del proyecto. En este momento se aprovecha para instalar todas las dependencias que serán utilizadas en el proyecto. La utilización de cada una se podrá ver más adelante.

Con el servidor inicializado y todas las dependencias instaladas, se puede empezar a desarrollar el servidor web. Para ello, hay que entender cuál es la estructura estándar de un servicio REST desarrollado en NodeJS. En la figura 15 se muestra esta estructura en forma de diagrama:

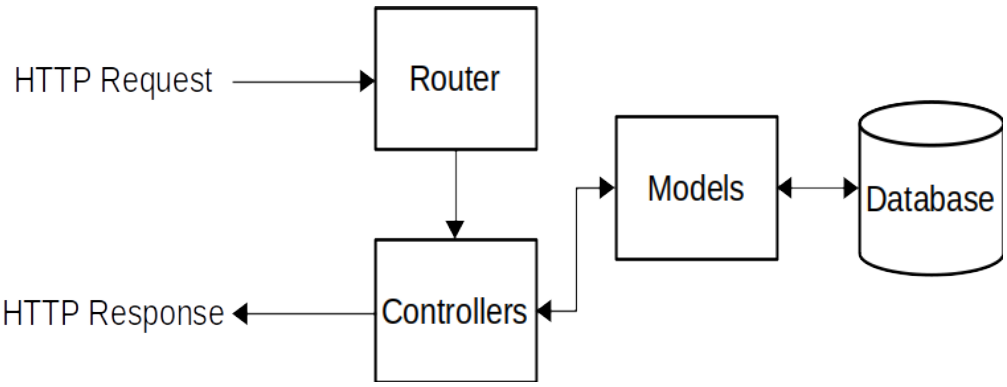


Figura 15: Estructura de un servidor REST en NodeJS

Como se puede ver, a parte de la base de datos, existen tres módulos muy diferenciados:

- **Enrutador:** Este módulo es el que se encarga de recibir las diferentes peticiones HTTP y de redireccionarlas al controlador correspondiente, dependiendo de los datos que estén siendo solicitados.
- **Controladores:** Los controladores se encargan de procesar la información que llega desde la petición HTTP y, con ella, realizar las operaciones necesarias en los modelos para poder devolver la respuesta HTTP solicitada por el cliente.
- **Modelos:** Los modelos son el equivalente en NodeJS a lo que en otros lenguajes de programación podríamos considerar objetos, pues existe un modelo que representa cada uno de los «tipos de datos» que existen en la base de datos.

En este proyecto dispondremos de un enrutador general, así como un controlador y un modelo para cada tabla de la base de datos. Además, se dispondrá de un controlador específico para la operación de login, puesto que de esta forma se puede encapsular más aún la operación que lleva los datos más sensibles en ella. Por el mismo motivo, habrá un modulo que únicamente almacenará las credenciales para el acceso a la base de datos y otro que almacenará las claves que se usan para el cifrado de contraseñas y tokens. Por último, existirá un fichero que se encargará de inicializar el servidor.

Además, para poder utilizar conexiones HTTPS, Godot requiere que la dirección a la que se conecta disponga de un certificado válido. Para conseguir uno, se ha usado la herramienta Certbot, propiedad de la autoridad de certificación Let's Encrypt. Certbot se utiliza a través de la terminal, y para la certificación de la dirección sólo es necesario seguir las instrucciones que muestra por pantalla. En un primer momento se introduce la dirección a certificar. Cuando Certbot verifica que la dirección existe, pide al usuario verificar que esa dirección le pertenece. Para ello, es necesario subir un fichero al servidor y hacerlo accesible a través de la propia dirección. Hecho esto, Certbot genera un par de archivos de certificación, el

certificado en sí y la clave secreta de éste. Estos dos archivos serán los que se usen para que la dirección web pueda ser accedida por medio de HTTPS, como más adelante se explicará.

Con esto, la estructura final de los archivos del servidor quedaría como se muestra en la imagen 16, en la que se puede observar la separación entre modelo, controladores y enrutador:

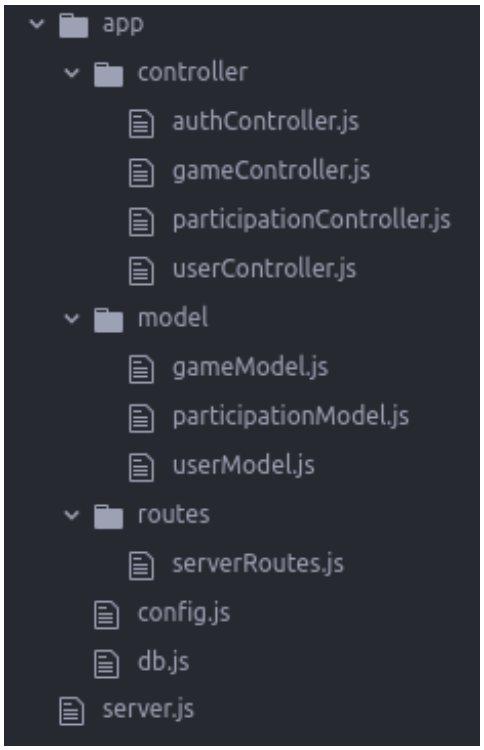


Figura 16: Estructura de los archivos del servidor

En primer lugar se crea el fichero «server.js», siendo este el que inicializa el servidor. Siguiendo el flujo de eventos del código, lo primero que se realiza es la lectura de los ficheros de los certificados del servidor, necesarios para poder utilizar el protocolo HTTPS. A continuación, se hace una redirección a todas aquellas conexiones que hayan accedido al servidor por medio de HTTP para que utilicen HTTPS. Después, mediante la librería «body-parser» se procesa la petición para poder tratar su cuerpo como un objeto JSON, lo que será útil cuando se necesite acceder a estos datos. En las siguientes líneas se ha implementado la redirección de la petición al módulo de enrutamiento, siendo este el que, tal y como se ha explicado anteriormente, procesará qué es lo que está siendo solicitado y realizará la petición correspondiente al controlador adecuado. En último lugar, el servidor es iniciado y queda a la espera de peticiones entrantes, tanto HTTP como HTTPS. Es en el momento de la creación del servidor HTTPS cuando son necesarios los certificados que se han obtenido anteriormente.

El siguiente módulo que se crea es el que lleva a cabo la conexión con la base de datos. Gracias a la librería MySQL, la conexión se puede realizar de una forma muy sencilla. Una

vez hecha, y en caso de que no haya habido ningún tipo de error al intentar conectarse, se exporta la conexión para que esta pueda ser importada y utilizada desde los módulos del modelo.

De un modo similar, se crea el módulo que contiene la información de las claves para el cifrado. La estructura es parecida al módulo de la base de datos, pero simplemente se exportan las variables que contienen dichas claves para que puedan ser utilizadas desde diferentes módulos.

A continuación, se deben crear los modelos. Todos los modelos que se van a realizar en este proyecto tienen una estructura interna similar: una constructora que dará al modelo los mismos campos que la tabla que representa en la base de datos, y unos métodos que ejecutan las funcionalidades necesarias para implementar todas las peticiones descritas en el diagrama de la estructura REST (figura 14).

En general, cada petición se realiza mediante un único método en el modelo correspondiente, pues la mayoría de peticiones se pueden resolver mediante dos pasos: en primer lugar, y sólo en caso de ser necesario, se comprueba si el usuario está proporcionando un token de autentificación; después se realiza una consulta en la base de datos y se devuelve como respuesta el resultado de esta consulta.

La única petición que necesita más de un método es la que crea una nueva partida. Ya que el usuario sólo proporciona el nombre de esta, es necesario que el servidor asigne el resto de parámetros. En el caso de el estado, se asigna el valor 0, pues inmediatamente después de crear la partida esta se encuentra en espera para que los jugadores se unan. En el caso del ganador, se asigna el valor -1, pues hasta que la partida no haya finalizado no tiene ningún sentido que haya un ganador. Sin embargo, para el puerto es necesario asignar uno que no se encuentre en uso. Es por eso que existe una función que comprueba en la base de datos los puertos que están en uso en el momento de la petición, y asigna a la nueva partida uno que no esté siendo utilizado.

Para la realización de los controladores, se necesita una estructura similar a la de los modelos. Es en estos en donde llegan las peticiones desde el enrutador, cada una llamando a un método diferente en función de la petición realizada. En estos métodos es donde, en caso de necesitarlo, se procesa los parámetros que llega junto a la petición. Por ejemplo, en el controlador de autentificación, es necesario decodificar las credenciales que envía el usuario en formato Base64. Después de que la información proporcionada sea procesada, se realiza la llamada al método necesario del modelo correspondiente. Siguiendo con el ejemplo de la autentificación, se llamará al método de login del modelo del usuario con el usuario y la contraseña decodificados como parámetros.

Por último se ha implementado el enrutador. Dado que en el diagrama de la estructura REST (figura 14) se definieron las rutas por las que llega cada una de las peticiones, y en el desarrollo de los controladores se han creado las funciones que procesan cada una de ellas, el trabajo del enrutador es interconectar estas. En él se asigna cada función a la ruta y método

necesario. Un ejemplo de esto se puede ver para la ruta «/games» en la imagen 17, en donde se redireccionan a diferentes funciones dos peticiones a la misma ruta, dependiendo si se hacen mediante GET o POST, para así llamar a la función que obtiene una lista de usuarios o a aquella que crea un nuevo usuario respectivamente.

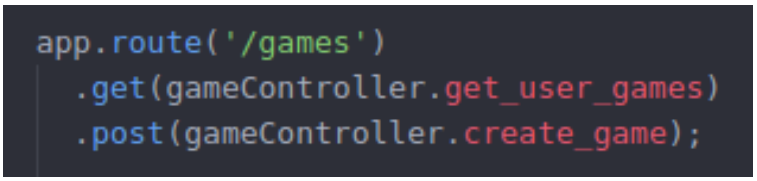


Figura 17: Código que redirecciona las peticiones a la ruta /games

Con esto, el desarrollo del primer prototipo queda terminado. Es momento de realizar las pruebas necesarias para comprobar su correcto funcionamiento.

6.1.2. Pruebas

Si bien es posible comprobar el correcto funcionamiento del servidor realizando diferentes peticiones HTTP «a mano», para la realización de las pruebas de la API REST del servidor se ha utilizado Postman. Esto es debido a que mediante el uso de esta herramienta se pueden definir todas las pruebas unitarias necesarias para comprobar todas las funcionalidades del servidor, pudiendo a continuación ser ejecutadas en serie para así comprobar en una única ejecución si todo funciona correctamente. Pese a que en un principio esto pueda parecer una pérdida de tiempo, pues es necesario invertirlo en desarrollar el código de las pruebas, a la larga se hace beneficioso al poder ejecutar todas las pruebas con un solo clic. Además, en caso de que en un futuro prototipo se vea la necesidad de modificar alguna parte del servidor, las pruebas ahora definidas son completamente reutilizables, por lo que después de la modificación se podría comprobar si todo sigue funcionando en unos pocos segundos.

En primer lugar, y para simplificar el desarrollo de las pruebas, se han creado las variables globales que representan las dirección a las que se harán peticiones, la cabecera «HTTP», y las diferentes rutas. Con esto, se pueden definir diferentes direcciones utilizando estas variables. Éstas se muestran en la figura 18.

VARIABLE	INITIAL VALUE
http	https
url	brawlchemy.info
api	api/v1
auth	auth
users	users
play	play
games	games

Figura 18: Variables globales definidas en Postman

Para comprender la utilidad de estas variables, se supondrá que se quiere definir una prueba unitaria que acceda a la dirección que comprueba el login de un usuario. Sin utilizar las variables definidas anteriormente, la URL a la que se hace la petición sería la siguiente:

```
https://brawlchemy.info/api/v1/auth
```

Sin embargo, al definir las variables, se puede realizar mediante la siguiente línea de texto, en la que se referencian las variables mediante dobles llaves:

```
{{http}}://{{url}}/{{api}}/{{auth}}
```

Gracias a esto todas las direcciones dependen del mismo conjunto de variables, por lo que en caso de ser necesario modificar una parte de la dirección se podría conseguir modificando únicamente la variable necesaria. Otra ejemplo de su utilidad sería modificar la variable «http» para cambiar entre HTTP y HTTPS, para comprobar que ambos estándares están soportados.

A continuación se han creado las pruebas unitarias. En el caso de este proyecto, para todas las rutas se ha definido una prueba que comprueba que el código de la respuesta es 200, código que comunica que la operación se ha realizado correctamente. Además, para las peticiones que esperan una respuesta con datos (como la operación de login, que espera recibir un token) se ha definido una prueba que comprueba que se recibe un dato correcto, además de la comprobación del código de la respuesta.

Estas pruebas deben seguir un orden definido para que se puedan comprobar correctamente todas las posibles funcionalidades. Éste se explica a continuación:

- En primer lugar se crea un nuevo usuario. Sus credenciales se generan de forma aleatoria al comienzo de la batería de tests y se guardan en unas variables globales que comienzan vacías.
- Tras registrar al usuario, se procede al login con sus mismas credenciales. Esta operación espera recibir un token como respuesta, el cual se almacena de la misma forma en una variable global en caso de ser recibido, puesto que será necesario en el resto de peticiones.
- A continuación se crea una nueva partida. El nombre de ésta se crea aleatoriamente.
- Después de crear la partida, se comprueba que esta aparece en las partidas disponibles para ser jugadas.
- La partida recién creada se actualiza para que quede marcada como finalizada.
- Tras finalizar la partida, se añade la participación del usuario actual en la misma.
- Después de añadir la participación, se pide al servidor que devuelva las participaciones en dicha partida. La que se acaba de añadir debería ser devuelta.
- Por último, se pide al servidor que devuelva las partidas que ha jugado el usuario. En caso correcto, sólo debería devolver la partida que se acaba de crear.

En varias de las pruebas es necesario definir parámetros de forma aleatoria. En postman esto se puede realizar mediante lo que se llama «pre-request script», un script en lenguaje JavaScript que se ejecuta antes de realizar las peticiones. Puesto que la primera petición necesita generar unos datos de usuario válidos pero diferentes a los ya existentes (ya que no pueden existir dos usuarios con las mismas credenciales), se utiliza la fecha en formato Unix para generar nuevos datos. Esto no es «completamente aleatorio», puesto que depende del momento en que se realizan los tests. Sin embargo, al crear los usuarios en base al momento actual, es seguro que no se repetirán credenciales al realizar las pruebas, cosa que aunque difícil, sí que podría ocurrir con valores puramente aleatorios. El código que hace posible esto se muestra en la figura 19:

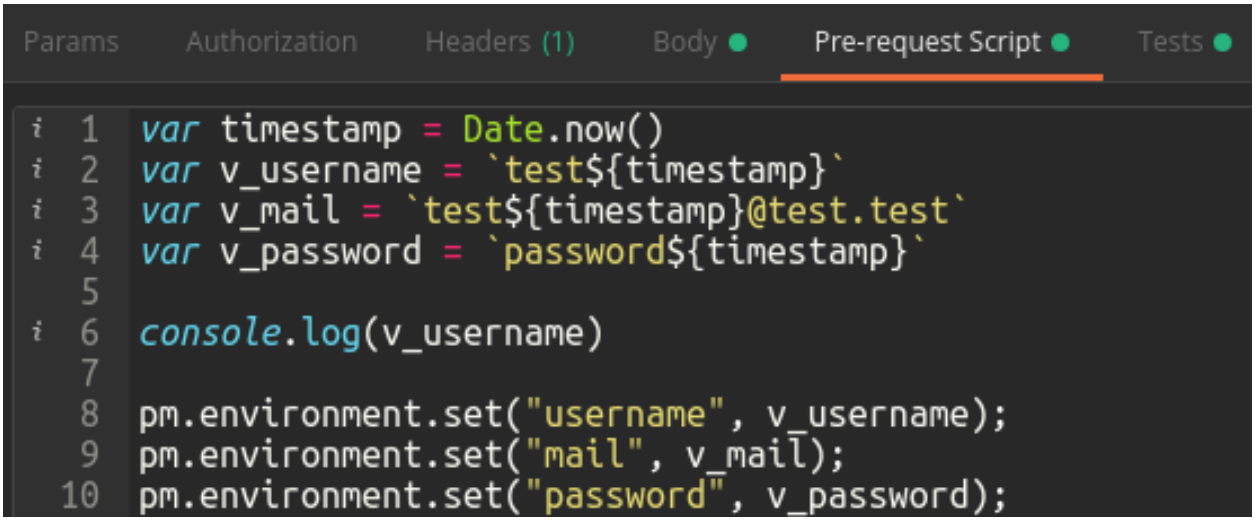


Figura 19: Generación de nuevas credenciales de usuario

En el bloque de código se puede ver que en primer lugar se obtiene el momento actual (variable timestamp). Éste se añade como texto en una serie de variables base para formar las credenciales del usuario final (por ejemplo: «test» + «23190913» + «@test.test» = «test23190913@test.test») y, en último lugar, se almacenan éstas en tres variables globales para su uso posterior.

Para explicar la estructura de una de las pruebas unitarias, se usará como ejemplo el inicio de sesión. La dirección a la que se hace la petición queda configurada en Postman como se muestra en la imagen 20. En ella se puede observar que en la mismas línea en la que se define la dirección a la que se hace la petición, también se define el tipo de petición que se realiza (en este caso GET).

Figura 20: Método y dirección de la petición del inicio de sesión

Teniendo en cuenta que esta prueba se ejecuta después de la de registro, se dispone de las variables globales que se han mostrado en la figura 19. El login se realiza mediante «Basic Auth», y Postman ofrece una forma de preparar esta autentificación automáticamente añadiendo los parámetros de usuario y contraseña en una de las opciones de la petición. Será el propio programa el que se encargue de realizar la codificación de estas credenciales a Base64 y añadir la cabecera necesaria en la petición automáticamente. Este menú, con las variables ya introducidas, se muestra en la imagen 21. En ésta se puede comprobar de nuevo cómo Postman permite usar las variables que se han definido en una prueba unitaria anterior, con lo que queda clara su gran utilidad de cara a la realización de pruebas en cadena.

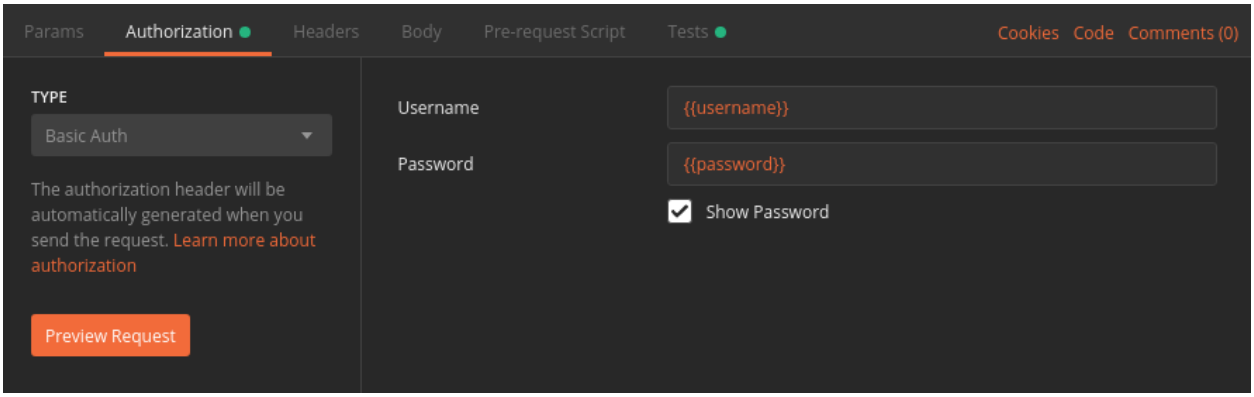


Figura 21: Introduccion de las credenciales necesarias para Basic Auth

Con la petición ya definida, es momento de desarrollar el código de los test. En el caso de esta petición, tal como se explicó anteriormente en esta sección, se debe comprobar tanto el código de respuesta de la petición como que se devuelva un token. Además, este token debe ser almacenado para las peticiones que siguen a ésta en la batería de tests, de igual forma que en la petición anterior fueron almacenados los datos de registro. El código se muestra en la figura 22.

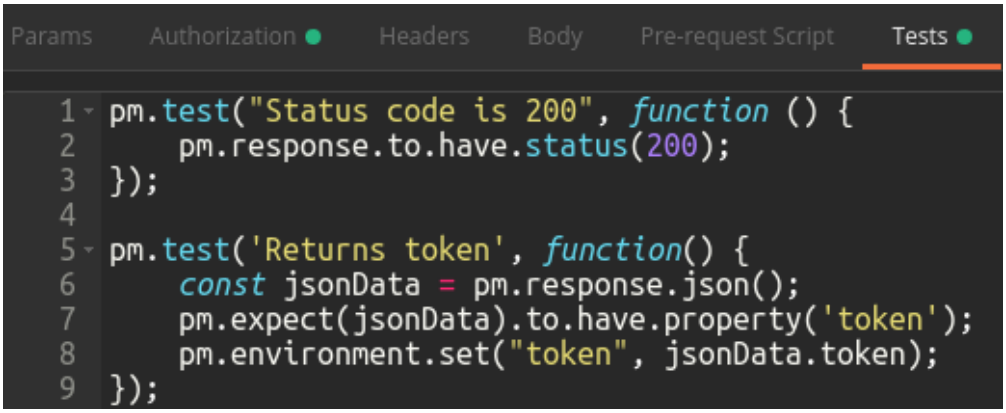


Figura 22: Tests del inicio de sesión

Los resultados obtenidos tras la realización de la primera ronda de pruebas se pueden ver en la tabla 42:

Tabla 42: Resultado de las pruebas del servidor

Prueba	**Resultado**
Registro de un nuevo usuario	OK
Inicio de sesión	OK
Creación de nueva partida	OK
Partida disponible para jugar	OK
Finalización de la partida	OK
Añadir participación del usuario	ERROR
Obtener participaciones	ERROR
Listado de las partidas jugadas	ERROR

Como se puede ver, las primeras pruebas se han realizado con éxito en sus resultados. Sin embargo, en la prueba para añadir participaciones ha ocurrido un error. Analizando la respuesta, esta nos indica que no ha sido posible registrar una nueva participación porque no se han enviado todos los parámetros necesarios. Comprobando la petición en Postman, estos sí que han sido enviados, por lo que el problema debe estar en la parte del servidor. En cuanto a los otros errores, se deben a que aunque se ha obtenido una respuesta del servidor, al no haberse podido registrar la participación en primer lugar, esta no aparece en las participaciones de la partida, así como tampoco aparece la partida entre las que el usuario ha jugado. Por ello, es posible que los errores en estas dos últimas se traten de fallos en cadena por el primer error.

Tras comprobar el código se encuentra que el error viene dado por la forma de comprobar la presencia de los parámetros. En el momento del fallo, la comprobación se realiza de la siguiente forma:

```
// Comprobar si param existe
if (!param) {
  // Devolver error por falta de parámetro
} else {
  // Procesar petición
}
```

Esto funciona en la mayoría de los casos, puesto que en JavaScript un parámetro nulo se considera falso, por lo que en caso de serlo se devuelve el error. Sin embargo, JavaScript también considera que el valor 0 es nulo. En el momento del fallo, se estaba tratando de registrar una participación en la que el usuario pertenecía al equipo azul, representado por

un valor 0. Al comprobar este parámetro, era considerado nulo y se devolvía un falso error. Esto se ha solucionado cambiando el código a una estructura como la que se puede ver en el siguiente bloque:

```
// Comprobar si param existe
if (param == null) {
  // Devolver error por falta de parámetro
} else {
  // Procesar petición
}
```

Tras este cambio se vuelven a realizar las pruebas. Los resultados de esta segunda batería se muestran en la 43:

Tabla 43: Resultado de las pruebas del servidor

Prueba	**Resultado**
Registro de un nuevo usuario	OK
Inicio de sesión	OK
Creación de nueva partida	OK
Partida disponible para jugar	OK
Finalización de la partida	OK
Añadir participación del usuario	OK
Obtener participaciones	OK
Listado de las partidas jugadas	OK

En esta ocasión todas las pruebas son superadas correctamente. Con esto, se puede considerar el primer prototipo como terminado y funcional.

6.2. Menús, login y registro

Durante el desarrollo de este prototipo se implementarán los menús del juego, entre los que se encontrará la funcionalidad de iniciar sesión en el servidor o registrarse en este, así como la posibilidad de crear partidas o unirse a éstas.

6.2.1. Desarrollo

Para desarrollar todas las funcionalidades de los menús del juego serán implementadas 6 escenas diferentes:

- Escena de registro e inicio de sesión,
- Escena que contenga el menú principal.
- Escena que muestre las partidas disponibles.
- Escena que contenga el lobby de la partida.
- Escena que muestre las partidas jugadas.
- Escena que muestre los controles.

6.2.1.1 Registro e inicio de sesión

Para realizar esta escena, será dividida en dos partes: puesto que no tiene sentido estar realizando la operación de registro a la vez que el inicio de sesión, mientras se realice una de ellas se ocultarán los elementos de la otra y viceversa.

Ambas partes comparten ciertos elementos, los cuales serán reutilizados en otras escenas manteniendo la misma funcionalidad. Estos serán referenciados a lo largo del desarrollo según el nombre que se les da a continuación:

- Un background, la imagen que sirve como fondo de pantalla para los menús,
- Un spinner, pequeño icono giratorio que aparecerá cuando el juego se encuentre en un proceso de carga (como cuando se espera una respuesta por parte del servidor).
- Una «cortinilla» que aparecerá para suavizar las transiciones entre escenas.
- Un popup que se mostrará en caso de que no haya podido establecerse la conexión con el servidor.

La estructura de la escena, sin haber añadido ninguna funcionalidad dentro de la parte de registro o login, queda como se muestra en la figura 23:

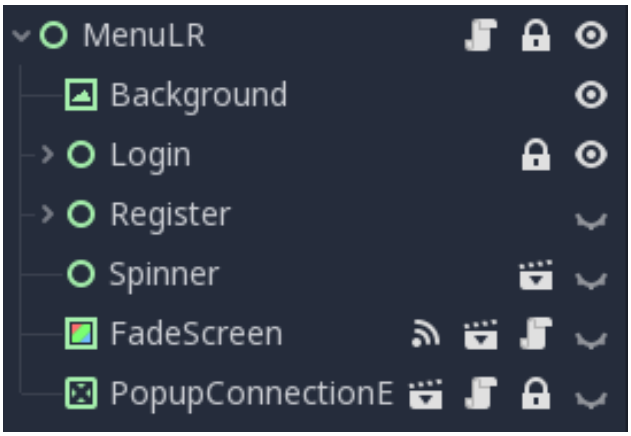


Figura 23: Estructura general de la escena de login y registro

Como se puede ver, existen dos nodos que contendrán los elementos pertenecientes al inicio de sesión y al registro respectivamente. Además, el nodo registro aparece oculto (denotado por el ojo cerrado), puesto que al utilizar esta escena se quiere que por defecto aparezca el menú de inicio de sesión. Del mismo modo tanto el spinner, la cortinilla y el popup se mantienen ocultos hasta que estos sean necesarios.

En cuanto al desarrollo de la parte gráfica del login y registro, ambos son muy similares, pues su estructura es también similar, al estar compuestos de una serie de campos en los que introducir datos y unos botones en la parte inferior con los que confirmar la introducción de datos o cambiar entre la pantalla de inicio de sesión y la de registro.

Para conseguir éste resultado se utilizan una serie de nodos llamados contenedores. Estos organizan los nodos que se encuentren como hijos en función del tipo que sean. Por ejemplo, un contenedor vertical ubicará sus hijos de forma vertical en la pantalla, de arriba hacia abajo uno tras otro. Uno horizontal, de la misma forma, lo hará de izquierda a derecha. Un tercer tipo es el contenedor central, que ubica sus hijos en el centro del mismo. Éste último no es utilizado en esta escena, pero en subsecuentes escenas será de utilidad.

Teniendo en cuenta que los contenedores pueden ser anidados, es posible diseñar menús que se mantengan consistentes sin importar el tamaño o la resolución de la pantalla en que se muestren, puesto que serán ellos los que se encarguen de ubicar los elementos en los lugares que se desee de forma consistente.

En las figuras 24 y 25 se puede ver la estructura final cada uno de los nodos.

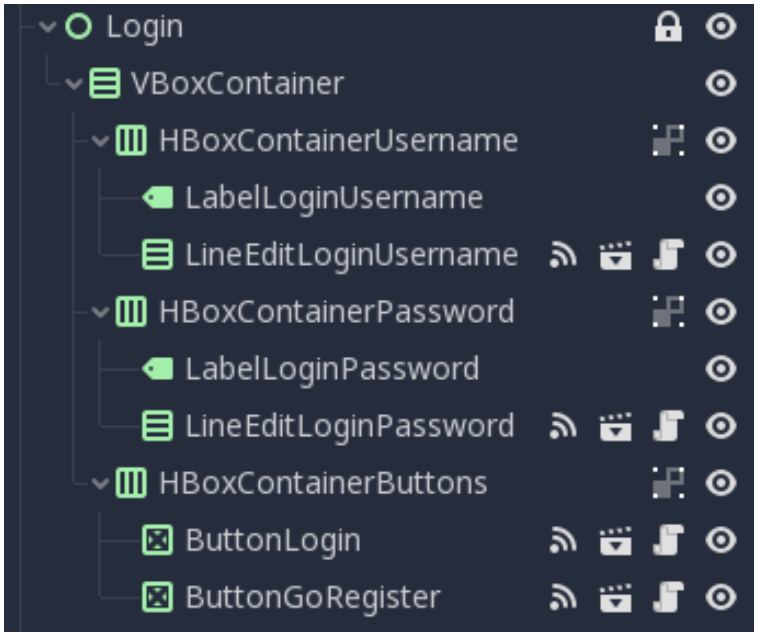


Figura 24: Estructura del nodo de login

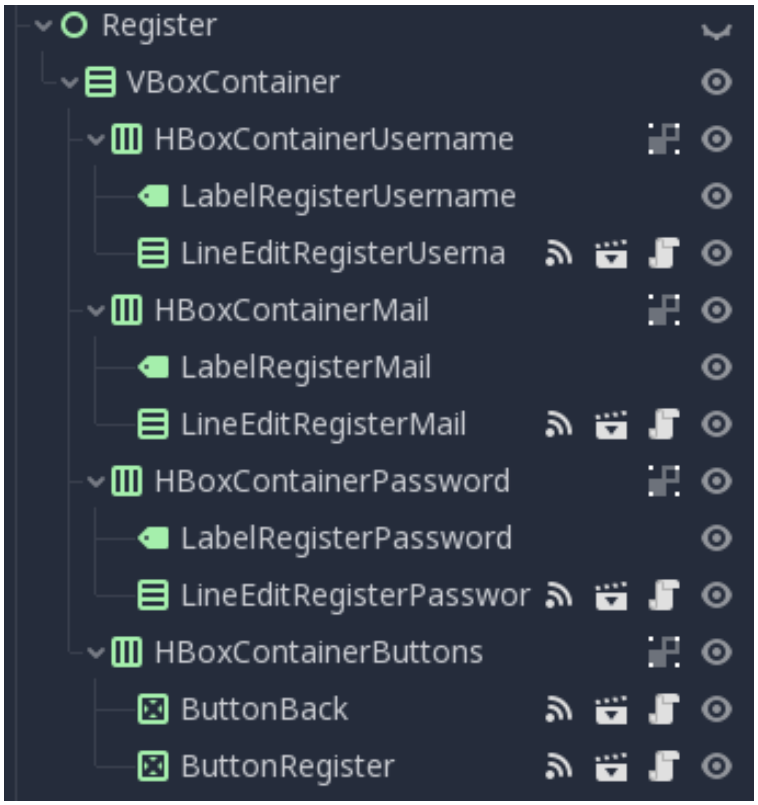


Figura 25: Estructura del nodo de registro

Estas estructuras se pueden entender más fácilmente si se ve junto al resultado que producen. La pantalla de inicio de sesión y la de registro se muestran en las imágenes

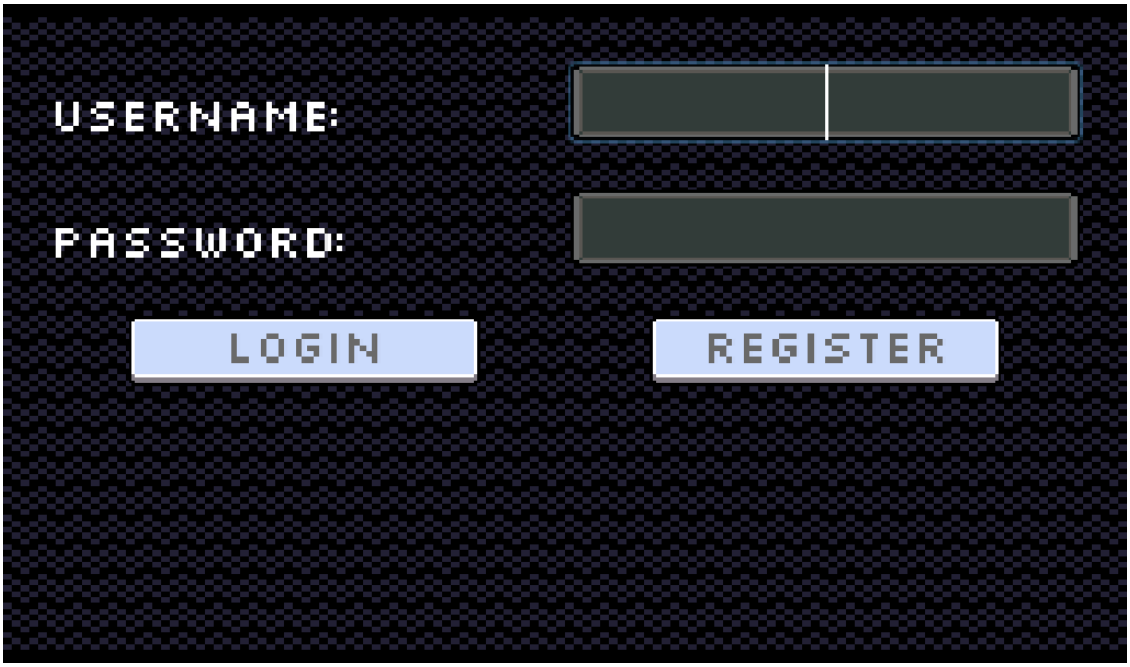


Figura 26: Pantalla de login

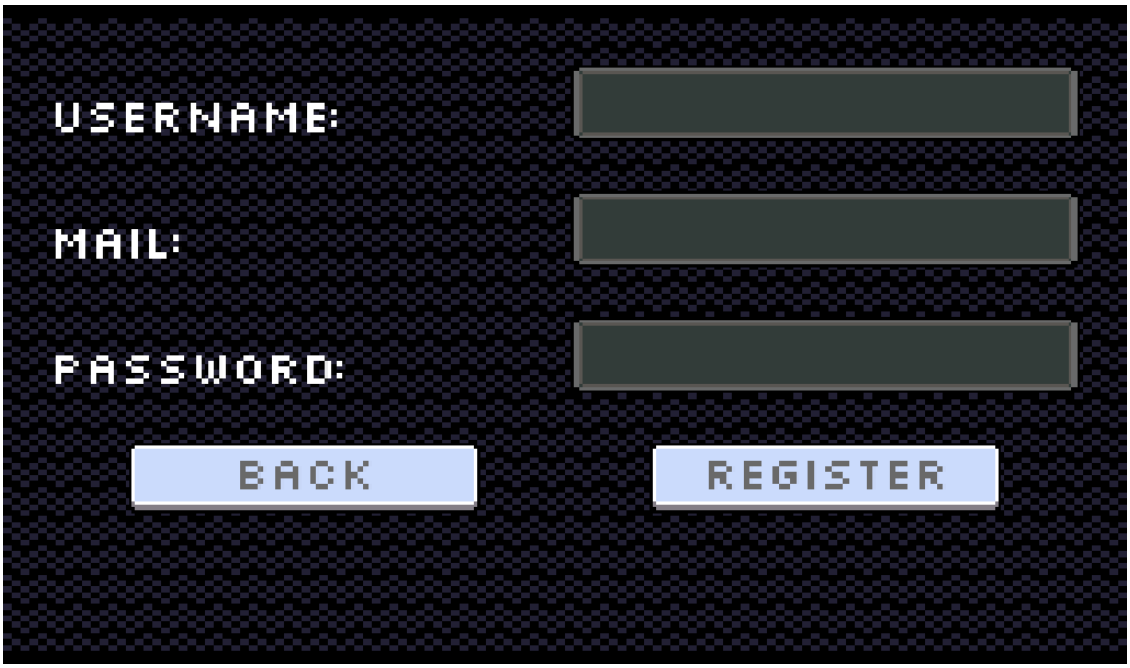


Figura 27: Pantalla de registro

Una vez desarrolladas las pantallas, se ha de añadir el script que implemente las funcionalidades necesarias:

- El botón «register» de la pantalla de inicio de sesión lleva a la pantalla de registro, así como el botón back de la pantalla de registro lleva a la pantalla de inicio de sesión.
- El botón «login» recoge los datos de los campos de inicio de sesión, los codifica en Base64 y los envía al servidor en un intento de inicio de sesión. En caso de que el servidor valide el login, el usuario es dirigido al menú principal. En caso contrario se muestra un mensaje de error en el campo del nombre de usuario.
- El botón «register» de la pantalla de registro recoge los datos de los campos del registro y comprueba que sean válidos en función de varios parámetros. Por ejemplo, la contraseña debe tener más de 8 caracteres, y la dirección de correo electrónico tiene que tener un formato válido. En caso de que todos los campos sean validados correctamente, se envían al servidor para dar de alta un nuevo usuario. En caso de que el servidor valide el registro, el usuario es redirigido a la pantalla de inicio de sesión. En caso contrario se muestra un mensaje de error en el campo que haya entrado en conflicto en el servidor.

La principal dificultad que se ha encontrado durante el desarrollo de esas funcionalidades ha sido el mostrar los errores. Muchos lenguajes de programación disponen de campos de texto a los que se puede asignar un texto de error, con lo que automáticamente se muestran en un color diferente, usualmente rojo, y muestran un texto indicando el tipo de error. En Godot esto no existe, por lo que para conseguir un resultado similar se ha implementado una escena personalizada que tiene esta funcionalidad. Esto se ha logrado mediante el uso de un campo de edición de texto y una pequeña etiqueta bajo éste (englobados en un contenedor vertical). Mediante un pequeño script, se permite asignar un mensaje de error a esta etiqueta cuando sea necesario, apareciendo éste en rojo. La estructura de esta escena y un ejemplo de un campo de texto con un error se pueden ver en las imágenes 28 y 29.

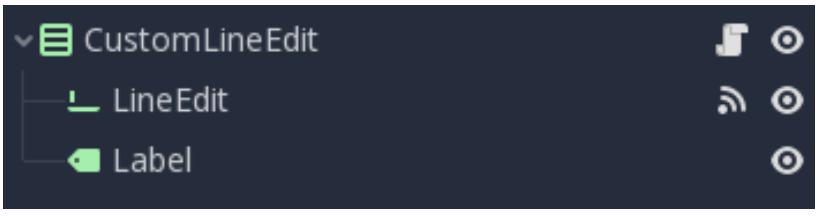


Figura 28: Estructura de la escena

Figura 29: Ejemplo de mensaje de error

6.2.1.2 Menú principal

El menú principal sirve como enlace a todas las funcionalidades disponibles para los usuarios identificados. Desde él tienen la posibilidad de jugar una partida, ver información de las partidas que han jugado en el pasado, ver los controles del jugador, o simplemente cerrar el juego. Es una escena sencilla de desarrollar al no tener funcionalidades complejas.

Para poder ir a las diferentes partes del juego se utilizarán una serie de botones que enviarán al usuario a la escena correspondiente. A parte de esto, un logotipo de la aplicación aparecerá en la parte superior del menú. Estos componentes estarán contenidos en un contenedor vertical. Además, igual que en otras escenas, se ha utilizado un background, una cortinilla y un spinner.

La estructura de la escena y el resultado final se pueden ver en las imágenes 30 y 31.

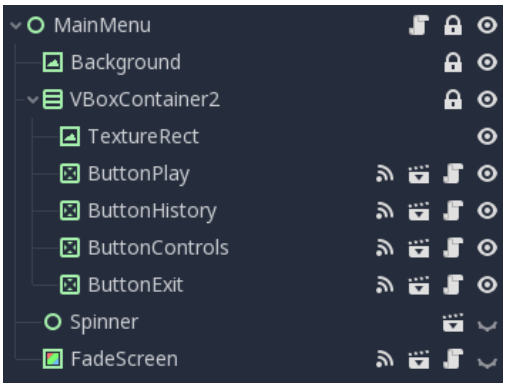


Figura 30: Estructura final de la escena del menú principal

Figura 31: Menú principal

6.2.1.3 Partidas disponibles

En la lista de partidas es donde el usuario podrá ver las diferentes partidas disponibles para jugar, seleccionar la que elija y unirse a ella. Además, se le dará la opción de crear una nueva partida en caso de que lo desee.

Sin embargo, para desarrollar la pantalla que lista las partidas disponibles es necesaria, en primer lugar, una forma de mostrar individualmente cada una de las partidas. Para ello se debe crear una escena personalizada que servirá como representación de cada partida de la lista. En esta escena se tendrá que mostrar el nombre de la partida, y deberá disponer de una función que avise al sistema de que esta partida ha sido seleccionada, pues para unirse ha ella el usuario ha de hacer clic encima de la misma. Además, esta escena deberá disponer del puerto en que se ubica la partida a la que represente, pues éste es necesario para realizar la conexión en caso de que la partida sea seleccionada. Por esto se ha escogido utilizar un botón como base de la escena, siendo éste modificado para actuar como una «etiqueta» que se adaptará adecuadamente al formato de lista en el que se va a incluir. El resultado de esta etiqueta se puede ver en la figura 32

Figura 32: Etiqueta que muestra una partida disponible

Disponiendo de una forma de representar las partidas, es posible implementar la escena que las liste. Para ello, de nuevo se ha recurrido al uso de contenedores. En este caso, dentro del contenedor principal se dispone de dos elementos: el que contiene la lista de las partidas y un segundo contenedor que contiene los botones que servirán para crear una partida, recargar la lista de partidas o volver al menú principal. Igual que en otras escenas, se han vuelto a utilizar el spinner, la cortinilla, el background y el popup que indica que no hay conexión.

Para la inicialización de la lista, se debe hacer una petición HTTP al servidor para que éste devuelva las partidas disponibles. Cuando se dispone de ellas, se recorren y se inicializa una etiqueta a la que se asigna el nombre de cada partida y su puerto correspondiente. Todas las etiquetas son añadidas consecutivamente al contenedor que las contendrá. En este caso se ha utilizado un «ScrollContainer», que trae por defecto implementada la funcionalidad necesaria para realizar un scroll en caso de que todas las partidas no quepan en la pantalla.

Por otro lado, en la parte inferior se encuentran los botones con las diferentes acciones que el usuario puede tomar. El último de ellos sirve para volver al menú principal, y su funcionalidad se basa en cambiar la escena a la del propio menú. Por otro lado, el botón de recargar fuerza al sistema a volver a enviar la petición que se mandó al cargar la escena, recibiendo de nuevo la lista de partidas disponibles. En el momento en que la recibe, elimina todas las partidas que se encuentren en el contenedor de la lista y, de igual forma que se hace

en un primer momento, añade consecutivamente las partidas que ha recibido como respuesta.

Por último se encuentra el botón de crear partida, teniendo éste más funcionalidad que los otros dos. Puesto que para crear una partida es necesario que el usuario de a esta un nombre, se necesita también una forma de que el usuario provea ese nombre. Para ello se ha creado un menú popup, similar al que se muestra cuando se da un fallo de conexión, en el que se puede introducir el nombre para la nueva partida y confirmar la operación, en cuyo caso se hará la petición HTTP correspondiente y, cuando se reciba una respuesta afirmativa del servidor, se recargará la lista de partidas automáticamente, apareciendo en ella la partida recién creada. Este menú popup se muestra en la figura 33.

Figura 33: Popup para la creación de una partida

El resultado final de la escena con unas partidas de prueba se puede ver en la imagen 34.

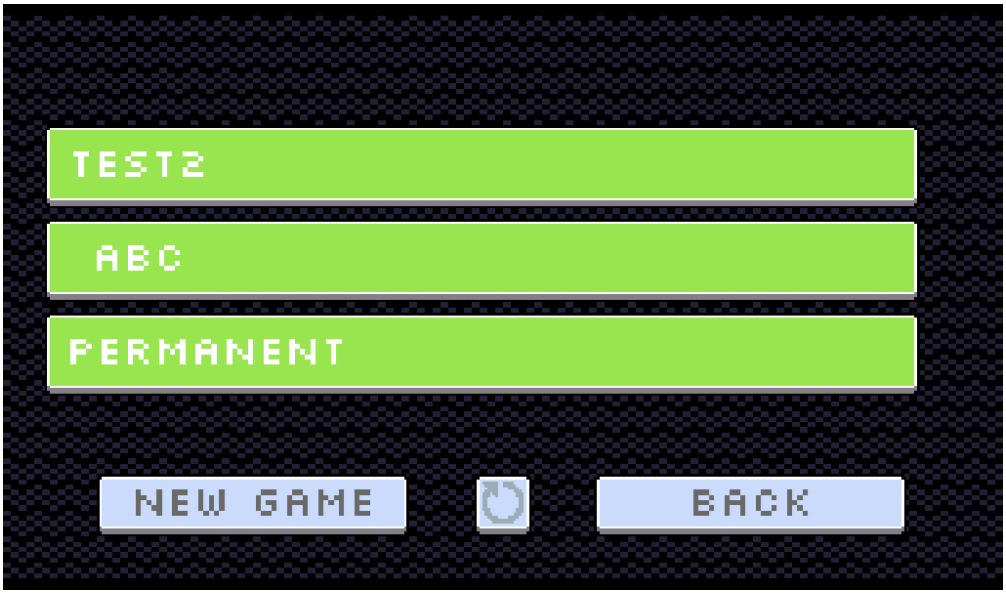


Figura 34: Menú de la lista de partidas

6.2.1.4 Lobby

La implementación de la escena del lobby es una de las más intrincadas del proyecto, puesto que contiene una gran cantidad de componentes y estos tienen que reaccionar a los eventos que provengan del servidor, como la conexión de nuevos jugadores, o el cambio de las propiedades de estos, como su equipo.

En primer lugar es necesario crear el script que servirá como servidor del juego, para lo que Godot dispone de un nodo llamado «NetworkedMultiplayerENet». Éste se crea durante la ejecución y es utilizado como punto de conexión entre los distintos usuarios de la red multijugador. Un aspecto importante es que uno de los nodos de la red será considerado el nodo principal (network master), siendo éste el primero en crearla y a través del cual el resto de nodos se comunicarán entre sí. Este nodo, el cual será llamado server de aquí en adelante, asigna una dirección IP a la que el resto de nodos deberán conectarse, generalmente y en el caso de este proyecto su IP local, así como un puerto a través del cuál se podrá acceder. Ya que se busca que sea la máquina virtual que gestiona el servidor web la que también gestione los distintos servers del juego, esta característica será especialmente útil para poder definir distintas instancias del servidor del juego en la misma IP pero escuchando en diferentes puertos. Como se ha comentado en apartados anteriores, en la base de datos se asigna a cada partida un puerto vacío, con lo que la asignación de éste es un problema ya resuelto.

En cuanto a la implementación de la NetworkedMultiplayerENet, a la que se llamará Network de aquí en adelante, Godot no ofrece más facilidades aparte de su inicialización como servidor o cliente, así como la conexión de un cliente a un servidor. Una vez se ha definido una red que incluya uno o varios clientes conectados a un servidor, la funcionalidad de la red queda en manos del servidor. Esta se lleva a cabo gracias a lo que ofrece la network: poder hacer llamadas remotas (RPC o *Remote Procedure Call*) a nodos que se encuentran en otra de las máquinas de la red, siempre y cuando la estructura de la escena de ambas máquinas (la que realiza la llamada y la que la recibe) sea la misma. Esta condición es la única que requiere Godot para que la network funcione, y el sentido de ella se puede ver fácilmente con el ejemplo de la figura 35 :

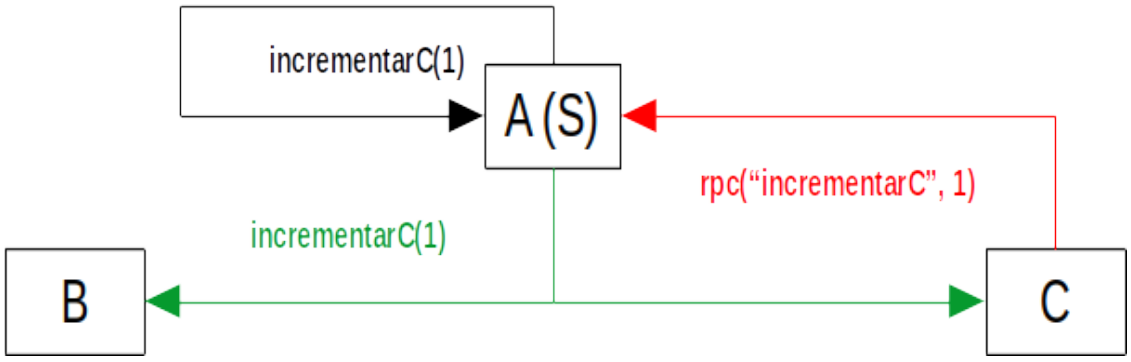


Figura 35: Ejemplo del funcionamiento de una RPC

En la imagen se puede ver una red con 3 máquinas interconectadas: A (Server), B y C. Supongamos que en el juego que se está jugando, la máquina C ha conseguido sumar 1 punto en su marcador, siendo éste un nodo que está en la raíz de la escena en la que se encuentran las tres máquinas y el cual dispone de un método que incrementa la puntuación actual de C, «incrementarC». Para que C pueda comunicar al resto de máquinas que ha conseguido aumentar su puntuación, esta debe en primer lugar comunicar al servidor que quiere aumentar su puntuación (flecha roja). Cuando el servidor reciba esta petición, buscará el nodo en la raíz de su escena y llamará en sí misma al método que aumenta la puntuación del jugador C (flecha negra). A continuación, hará una llamada remota a todo el resto de máquinas de la red, indicando que deben llamar al mismo método para realizar la misma operación (flecha verde). De esta forma es el server el que realiza la lógica del juego, mientras que el resto de las máquinas de la red son lo que se conoce como «puppets» o marionetas, puesto que toda su funcionalidad viene dada por las acciones de otra máquina.

Con esto se puede explicar la razón de que la estructura de las escenas deba coincidir, ya que los métodos siempre son relativos a un nodo de las escenas y a su vez pueden actuar en otros nodos, con lo que si la máquina B tuviese el método para aumentar el marcador en un nodo ubicado en un lugar diferente de la raíz, no podrá realizar la llamada en la ruta que el server le ha comunicado, creando así una disparidad entre las distintas instancias de la red y, en el peor de los casos, pudiendo llegar a bloquear el juego o terminar su ejecución si el fallo se produce en el propio server.

El lobby es la primera escena del juego que necesitará utilizar esta network, pues es la primera en la que se «interactúa» con otros jugadores en tiempo real, y por tanto el server necesita inicializarse en esta escena para que los jugadores puedan acceder directamente al lobby de la nueva partida. El problema que surge en este momento es la forma de crear el server, puesto que debe hacerse de una forma que la máquina virtual sea capaz de iniciar directamente el server del juego sin pasar por los menús, siendo esto último lo que tiene que hacer un jugador normal con su registro, inicio de sesión y selección de partida. Para ello se ha creado una escena extra, que será la que se carga automáticamente al iniciar el juego. Ya que Godot permite iniciar juegos desde la terminal (siendo esta la forma de crearlos desde NodeJS), es posible indicarle parámetros. Mediante esta función será posible realizar una llamada como la que se hace en el siguiente comando:

```
godot --path /path/to/Brawlchemy --server --port 1234
```

En ella, indicamos a Godot que se quiere inicializar Brawlchemy (ubicado en el directorio que se pasa como parámetro tras «–path»), que éste sea una instancia de servidor, así como el puerto en que tiene que escuchar. En esta nueva escena que se acaba de crear se comprueban los parámetros que se han pasado: en caso de no existir ninguno, se dirige al usuario al menú de login, pues al iniciar el juego de forma normal no se utilizan parámetros; en caso de existir el parámetro «–server», se iniciará la network como servidor en el puerto indicado por el parámetro «–port XXXX» y el juego quedará a la espera de jugadores.

Ahora que el server es capaz de inicializarse directamente en el lobby, es necesario desarrollar la funcionalidad de la propia network para que reaccione a los jugadores que se conecten o desconecten de ésta. El nodo «NetworkedMultiplayerENet» proporciona una serie de señales que serán utilizadas para diferentes funciones:

- **connection_succeeded:** Se emite cuando la network local logra conectarse al servidor. Cuando esto ocurre, se ha creado un método que comunica al server la información del usuario que se acaba de unir (nombre, equipo, etc), y es éste el que la comunica al resto de jugadores.
- **connection_failed:** Se emite cuando la network local no es capaz de conectarse a la network del servidor, y se utiliza para mostrar un mensaje de error en caso de fallo al intentar conectarse al lobby.
- **server_disconnected:** Se emite cuando el server se desconecta. En caso de que esto ocurriese se debería a algún tipo de fallo por parte del server, pues no es una funcionalidad planeada, por lo que se ha gestionado de forma que los usuarios vuelvan al menú principal.
- **peer_connected:** Esta señal es similar a la primera, pero mientras que la primera sólo se emite en la máquina que se ha conectado, esta se emite en todos los nodos de la red cuando uno nuevo se conecta.
- **peer_disconnected:** Se emite cuando un nodo de la red se desconecta de esta. Es usada en el servidor, de modo que cuando un cliente es desconectado, se borra la información del jugador y se comunica el borrado al resto de nodos.

Aparte de estas señales, se han definido una serie de métodos para gestionar todos los cambios que puede realizar el jugador: de equipo (azul o rojo), de personaje y de estado (listo o no listo). Estos comunican al server la nueva información del usuario, y el server la comunica al resto de nodos de la red.

Además, siempre que haya algún cambio en los jugadores, el servidor comprobará si los requisitos para comenzar la partida se cumplen: 3 jugadores en cada equipo, estando todos ellos en estado listo. En caso afirmativo, el servidor realiza un cambio de escena a todos los jugadores conectados, llevándolos a la escena del escenario en que se jugará la partida. Esta escena se realizará en el siguiente prototipo.

Por último, es necesario poder detener la ejecución de los distintos servers de Brawlchemy en caso de que estos se encuentren inactivos, pues no tiene sentido mantener en uso los recursos de la maquina en la que corren cuando estos pueden ser liberados. Para ello, se ha creado un temporizador que se activa en el momento en que no haya ningún jugador en el lobby. Al cabo del tiempo establecido, en este caso 5 minutos, se procede a eliminar la partida disponible de la base de datos y el lobby. Para lo primero, se realiza una petición

HTTP en la que se actualiza el estado de la partida a -1, puesto que no ha sido jugada y va a dejar de estar disponible. A continuación, se manda la señal que proporciona Godot para detener por completo la ejecución, con lo que el server es completamente eliminado.

Pasando al apartado gráfico de la escena del lobby, se ha seguido un esquema basado en diferentes contenedores similar a los anteriores. Dentro del contenedor principal existirán tres filas: una para las listas de jugadores y el botón de cambio de equipo, otra para el icono del personaje elegido y los botones necesarios para cambiarlo, y una última para marcar el estado de listo o abandonar la sala.

Para la implementación de las listas de jugadores, se han usado dos contenedores verticales vacíos a los que se irán añadiendo los jugadores a medida que se unan a cada uno de los equipos. Estos jugadores estarán representados por una escena personalizada en forma de «etiqueta» que se muestra diferente en función de si el jugador es del equipo rojo o azul, de si es el usuario actual, y si ha marcado que está listo. La escena esta compuesta por un fondo que representa el equipo del jugador (azul o rojo) y si esta listo (denotado por un borde verde), y una etiqueta sobre éste en el que se escribe el nombre del jugador, siendo éste verde en caso de ser el usuario local. La asignación de las propiedades se hace mediante un script que es llamado antes de añadir el jugador a la lista y en el que se asignan los diferentes parámetros que deben ser representados. En la imagen 36 se puede ver un ejemplo de cómo se ven las diferentes etiquetas, siendo «test» el jugador local.

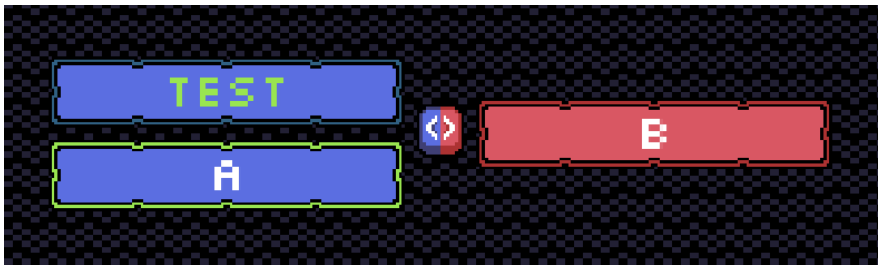


Figura 36: Distintas visualizaciones de los jugadores del lobby

En la imagen anterior se puede ver además el botón de cambio de equipo, que al ser pulsado comunica al servidor que se desea cambiar al equipo contrario, procesando éste los datos del jugador para actualizarlos adecuadamente y comunicar después este cambio al resto de jugadores.

El siguiente paso es reaccionar a las modificaciones de los jugadores que se encuentran en la sala. Para ello, se ha creado una señal en la network que se emite cuando un jugador informa de que sus propiedades han cambiado. Esta señal se conecta a un método en el script del lobby, siendo éste el que recibe la información actualizada de los jugadores en forma de JSON y recorre cada uno de ellos, asignando sus propiedades a cada etiqueta y finalmente añadiendo éstas al contenedor correspondiente.

Bajo las listas de jugadores se encuentra un nuevo contenedor horizontal que contendrá el

apartado para la selección del personaje (figura 37). En él se encuentran dos botones, y entre estos un icono que representa el personaje actualmente seleccionado. Estos botones están conectados a los métodos que anteriormente se crearon para esta función y, de igual forma que el botón de cambio de equipo, comunican al server el nuevo personaje seleccionado por el jugador, y es el server el que realiza el cambio y lo comunica al resto de jugadores.

Figura 37: Zona de selección de personaje

Por último, se han creado los botones de la parte inferior. El primero de ellos será el que se use para seleccionar el estado de listo, comunicándolo al servidor igual se hace con el cambio de equipo o personaje. Por otro lado, el segundo botón será utilizado para salir de la partida. éste se desconecta del server y después cambia la escena a la lista de partidas.

En la imagen 38 se muestra el resultado final del lobby.

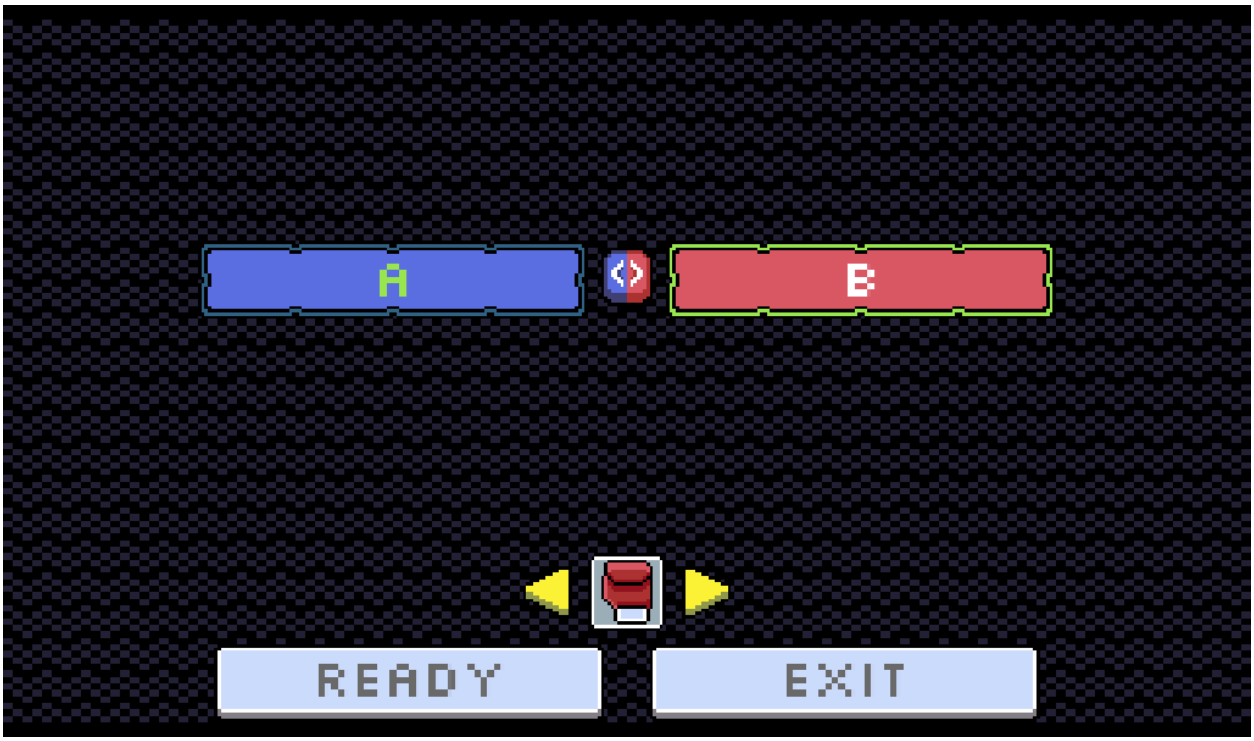


Figura 38: Lobby de la partida con dos jugadores en ella

6.2.1.5 Partidas jugadas

La escena del historial de partidas se ha dividido en dos subescenas: una de ellas muestra la lista de partidas jugadas y en otra se muestra la información de la partida seleccionada.

Para la realización de la primera de estas dos, se ha utilizado un método de desarrollo similar al que se usa en la lista de partidas disponibles, pues la estructura de la escena es muy similar al estar ambas formadas por una lista de elementos. Esta consiste en una serie de etiquetas representando a las partidas jugadas, así como un botón para volver al menú principal en la parte inferior, junto al ya mencionado spinner, la cortinilla y un popup que notifica de errores de conexión. En el momento de abrir la escena se hace la petición correspondiente al servidor, y en el momento de recibir las partidas jugadas se crea una etiqueta para cada una de ellas, siendo esta inicializada con el nombre de la partida y con un fondo personalizado en función del equipo ganador de ésta.

A diferencia de la escena de las partidas disponibles para jugar, en esta escena no es necesario disponer de un botón de recarga, ya que las partidas no se pueden ver modificadas en tiempo real (pues es necesario jugar para que aparezca una nueva partida en esta lista).

En la figura 39 se puede ver un ejemplo de una de estas etiquetas.

Figura 39: Etiqueta de partida jugada ganada por el equipo azul

Cuando se selecciona una de las partidas jugadas, se accede a la escena que muestra la información de ella. Para ello, se realiza una petición HTTP a la dirección definida en el servidor (ver figura 14), que devuelve un lista con las participaciones de la partida seleccionada. La escena, haciéndose uso de contenedores, dispone de dos columnas diferentes a las que añadir las estadísticas de la participación de cada jugador, siendo cada una utilizada para un equipo diferente. Estas estadísticas se representarán mediante una pequeña escena que en la que, actuando a modo de etiqueta, aparece el nombre de usuario, sus muertes y sus asesinatos. Para configurar cada una de ellas se dispone de un método que las inicializa con los datos proporcionados, tras lo cual pueden ser añadidas a su columna correspondiente. Además, en la parte superior de dicha columna se ha añadido un encabezado que facilita la comprensión de los datos mostrados en la escena.

En la figura 40 se muestra un ejemplo del resultado final de la escena.

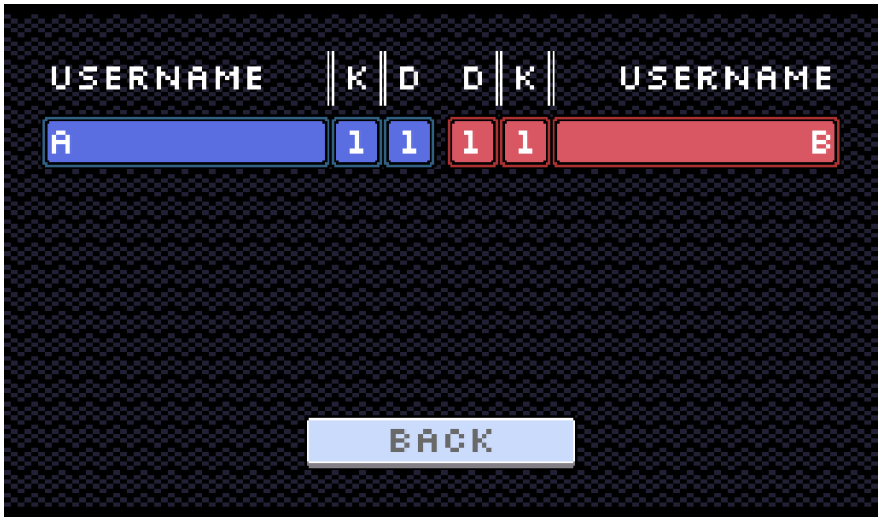


Figura 40: Menú que muestra las participaciones en una partida

6.2.1.6 Controles

Esta es la escena más simple de los menús, pues consiste únicamente en una imagen que muestra los controles y un botón que permite volver al menú principal.

Para desarrollarlo, se ha utilizado un contenedor vertical que contiene en primer lugar la imagen con los controles, seguido del botón para volver al menú principal. Aparte de esto, se utiliza el mismo fondo de pantalla que se ha utilizado en otros menús, así como la «cortinilla» que sirve de transición entre escenas. Además, para que el botón aparezca siempre en el final de la partida, se ha añadido un nodo vacío cuya finalidad es ocupar espacio y desplazar el botón hasta la parte de debajo para mantener una consistencia entre menús.

La estructura de la escena quedaría como se puede ver en la figura 41:

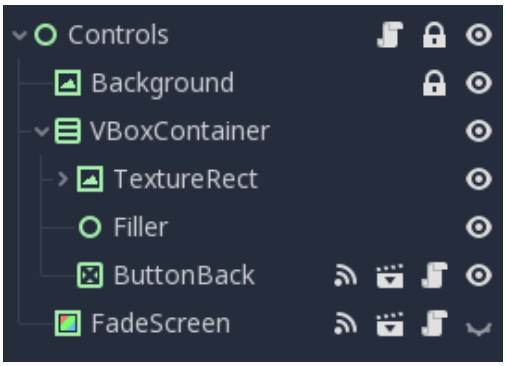


Figura 41: Estructura final de la escena de los controles

El resultado final de la pantalla de controles se puede ver en la imagen 42:

Figura 42: Menú que muestra los controles del juego

6.2.2. Pruebas

Para comprobar el funcionamiento de los menús se han definido una serie de pruebas para cada uno de ellos. En las tablas a continuación se muestran las pruebas definidas para cada menú, la descripción del resultado esperado, así como el resultado obtenido.

6.2.2.1 Pruebas del registro e inicio de sesión

A continuación, en la tabla 44, se muestran las pruebas realizadas sobre el menú de login y registro.

Tabla 44: Resultado de las pruebas del menú de login y registro

Prueba	**Resultado esperado**	**Resultado obtenido**
Inicio de sesión con un campo vacío	Se muestra un mensaje de error en el campo vacío.	OK
Inicio de sesión con datos erróneos	Se muestra un mensaje de error avisando de que no se ha podido iniciar sesión.	OK
Inicio de sesión con datos correctos	Se guarda el token y se cambia a la pantalla del menú principal.	OK
Ir al registro	Se cambia a la pantalla de registro.	OK
Registro con un campo vacío	Se muestra un mensaje de error en el campo vacío.	OK
Registro con datos no válidos	Se muestra un mensaje de error explicativo en el campo con datos no válidos.	OK
Registro con usuario o mail ya existentes	Se muestra un mensaje de error en el campo que crea conflicto.	ERROR 1
Registro con datos correctos	Se cambia a la pantalla de login.	OK
Ir al login	Se cambia a la pantalla de login.	OK

Resolución de errores:

- **ERROR 1:** Cuando se trata de realizar un registro con un email o usuario previamente registrado, no ocurre nada. A pesar de que esta situación estaba prevista en el servidor y existe una respuesta específica para ello, no se había desarrollado una reacción a ella por parte del juego. Este error se ha podido solucionar fácilmente, pues el código necesario para programar este mensaje es similar al que se usa cuando el inicio de sesión se realiza con datos incorrectos.

6.2.2.2 Pruebas del menú principal

A continuación, en la tabla 45, se muestran las pruebas realizadas para comprobar el correcto funcionamiento del menú principal.

Tabla 45: Resultado de las pruebas del menú principal

Prueba	**Resultado esperado**	**Resultado obtenido**
Ir a la lista de partidas disponibles	Se cambia a la pantalla de partidas disponibles.	OK
Ir a la lista de partidas jugadas	Se cambia a la pantalla de partidas jugadas.	OK
Ir al menú de controles	Se cambia a la pantalla de controles.	OK
Abandonar el juego	Se cierra la aplicación.	OK

6.2.2.3 Pruebas de la lista de partidas disponibles

A continuación, en la tabla 46, se muestran las pruebas realizadas para comprobar el correcto funcionamiento del menú de partidas disponibles.

Tabla 46: Resultado de las pruebas del menú de partidas disponibles

Prueba	**Resultado esperado**	**Resultado obtenido**
Visualizar automáticamente las partidas disponibles	Las partidas disponibles aparecen al entrar en el menú.	OK
Recargar las partidas disponibles	Las partidas disponibles son actualizadas.	OK
Abrir menú de creación de nueva partida	Aparece el menú de creación de partidas.	OK
Crear nueva partida	Se crea una nueva partida y se actualizan las partidas disponibles.	ERROR 2
Cerrar menú de creación de nueva partida	Desaparece el menú de creación de partidas.	OK
Unirse a partida	Se accede al lobby de la partida seleccionada.	OK
Ir al menú principal	Se cambia a la pantalla del menú principal.	OK

Resolución de errores:

- **ERROR 2:** Cuando se trata de crear una nueva partida, no ocurre nada. La causa encontrada es la lectura incorrecta del texto que introduce el usuario como nombre de partida, ya que se extraía de un campo incorrecto, y como consecuencia se enviaba un nombre vacío, siendo rechazado en el servidor. Este error se ha podido solucionar cambiando el campo del que se extrae dicho nombre.

6.2.2.4 Pruebas del lobby

A continuación, en la tabla 47, se muestran las pruebas realizadas para comprobar el correcto funcionamiento del lobby.

Tabla 47: Resultado de las pruebas del menú del lobby

Prueba	**Resultado esperado**	**Resultado obtenido**
Visualizar a todos los jugadores	Las jugadores de la partida aparecen y desaparecen correctamente.	ERROR 3
Visualizar estados	Se visualiza el estado (listo - no listo) de cada jugador de la partida.	OK
Cambiar estado	Se cambia el equipo del usuario.	OK
Visualizar equipos	Se visualiza el equipo (azul - rojo) de cada jugador de la partida .	OK
Cambiar equipo	Se cambia el equipo del usuario.	OK
Visualizar personaje	Se visualiza el personaje elegido por cada jugador de la partida.	OK
Cambiar personaje	Se cambia el personaje del usuario.	OK
Comenzar partida	Cuando todos los jugadores se encuentran listos, la partida comienza.	OK
Eliminar partida inactiva	Cuando la partida se encuentra inactiva durante un periodo de tiempo, esta se elimina del sistema.	OK
Ir al menú principal	Se abandona el lobby y se cambia a la pantalla del menú principal.	OK

Resolución de errores:

- **ERROR 3:** Cuando se trata de accede al lobby, se dan situaciones en las que los jugadores no aparecen representados hasta que estos se actualizan, ya sea porque se

une uno nuevo, se actualiza la información de alguno de ellos, o uno abandona el lobby. Esto se debe a que se puede dar el caso de que el servidor envíe la información de los jugadores mucho antes de que la escena del lobby haya cargado, con lo que no puede reaccionar a la señal que envía el server. Por ello, se ha añadido una nueva función en la network para que un cliente pueda solicitar la información de los jugadores del server. Cuando el lobby termina de cargar, se hace una llamada a esta función para que sea en ese momento cuando el servidor envíe al cliente necesario toda esta información, pudiendo entonces mostrarse correctamente.

6.2.2.5 Pruebas de la lista de partidas jugadas

A continuación, en la tabla 48, se muestran las pruebas realizadas para comprobar el correcto funcionamiento del menú de las partidas jugadas.

Tabla 48: Resultado de las pruebas del menú de partidas jugadas

Prueba	**Resultado esperado**	**Resultado obtenido**
Visualizar automáticamente las partidas jugadas	Las partidas jugadas aparecen al entrar en el menú.	OK
Abrir menú de información adicional	Aparece el menú de información de la partida.	OK
Cerrar menú de información adicional	Desaparece el menú de información de la partida.	OK
Ir al menú principal	Se cambia a la pantalla del menú principal.	OK

6.2.2.6 Pruebas del menú de controles

A continuación, en la tabla 49, se muestran las pruebas realizadas para comprobar el correcto funcionamiento del menú de controles.

Tabla 49: Resultado de las pruebas del menú de controles

Prueba	**Resultado esperado**	**Resultado obtenido**
Visualizar los controles	Aparecen todos los controles del juego.	OK
Ir al menú principal	Se cambia a la pantalla del menú principal.	OK

6.3. Creación de los personajes jugables

Durante el desarrollo de este prototipo se implementarán los personajes jugables y el entorno en el que se ubicarán, junto a las interacciones entre ellos.

6.3.1. Desarrollo

Antes de comenzar a desarrollar los personajes, las estructuras defensivas y el mapa en el que se ubicarán, es necesario establecer una serie de parámetros para ellos desde el primer momento, puesto que en pasos siguientes será necesario que estos personajes interactúen entre sí.

En primer lugar se definirán las capas (layers) y las máscaras (masks) para todos los nodos. En Godot, cuando se trabaja en un entorno 2D, se pueden definir estos dos valores para los nodos que tienen una representación «física» en la escena, como puede ser un personaje o el mapa. La capa hace referencia al plano o planos en que un nodo y sus hijos estás ubicados, mientras que la máscara son aquellos planos con los que hará comprobaciones para gestionar colisiones y otras interacciones. Existe un total de 20 posibles planos, estando por defecto numerados pero pudiendo ser nombrados por el desarrollador para comprender mejor los nodos que se encontrarán en él.

Por ejemplo, en un juego con varios personajes aliados y varios enemigos, podríamos definir que los personajes aliados tengan como capa y máscara el plano 1, y los enemigos el plano 2. Con esto, los personajes aliados chocarían entre sí al moverse, pero no serían capaces de detectar a los enemigos, y lo contrario con los enemigos. Sin embargo, si definimos a los personajes aliados con capa 1 y máscara 2, y a los enemigos con capa 2 y máscara 1, los personajes aliados podrían interactuar con los enemigos y viceversa, pero ninguno podía interactuar con los de su mismo tipo.

Esta funcionalidad es especialmente útil para gestionar las colisiones entre personajes y entorno, pues por ejemplo no se desea que los personajes atraviesen el suelo, así como la detección de los ataques, pues en este juego un jugador no debería ser capaz de atacar a otro de su mismo equipo. De esta forma se pueden programar las colisiones de una forma general, y utilizar las capas para definir qué colisiona con qué, en vez de tener que comprobar en el código si la colisión se está dando con un objeto dado y después, en caso de ser necesario, actuar en consecuencia.

En Brawlchemy existen 3 tipos de nodo diferentes para los que habrá que gestionar las capas de forma distinta:

- **Terreno:** Estos nodos serán de los que está compuesto el mapa del juego. Todos los jugadores, sean del equipo que sean, deben poder colisionar con el terreno para evitar atravesarlo.

- **Equipo azul:** A este tipo pertenecerán tanto los personajes que controlan los jugadores del equipo azul, como las estructuras de dicho equipo. Estos personajes no deben poder interactuar entre ellos, pero sí deben poder interactuar con los jugadores y estructuras del equipo rojo.

- **Equipo rojo:** Del mismo modo que el equipo azul, el equipo rojo no deberá poder interactuar con nodos del equipo aliado, pero sí con los del equipo enemigo.

Con estas especificaciones es posible definir las capas y máscaras de todos los equipos como se muestra en la tabla 50:

Tabla 50: Asignación de capas y máscaras

		Plano 1	Plano 2	Plano 3
Terreno	Capa	**X**		
	Máscara			
Equipo azul	Capa		**X**	
	Máscara	**X**		**X**
Equipo rojo	Capa			**X**
	Máscara	**X**	**X**	

Como se ha dicho antes, Godot permite dar un nombre personalizado a los planos para que sea más sencillo recordar lo que hacen. En vistas a la tabla anterior, tiene sentido nombrarlos en función del elemento que se encuentra en ellos, es decir, en base a la capa de los elementos. Por ello se llamará a los planos capa del terreno, capa del equipo azul y capa del equipo rojo respectivamente. Con estas asignaciones hechas, se puede comenzar a realizar la implementación.

6.3.1.1 Desarrollo del mapa

Lo primero que se desarrollará será la escena que contendrá el mapa. En un primer momento sólo contendrá el terreno de éste, puesto que tanto los jugadores como las estructuras defensivas de los equipos se añadirán más adelante tras ser implementadas en una escena aparte, ya que necesitan tener funcionalidad propia.

La base de la escena será un nodo simple sobre el que se ubicará un TileMap. Este nodo necesita que se le proporcione una imagen de un tileset y, tras definir cada tile en ella y darle un area de colisión, permite utilizar cada uno de los tiles individuales para realizar el mapa. Estos se deben colocar en la escena, como si se tratase de un mosaico, para generar el entorno deseado para el juego. En la imagen 43 se muestra la forma de definir los tiles y su area de colisión (en este caso un cuadrado que cubre el tile por completo); y en la imagen

44 se puede ver como se ha usado el TileMap para crear una pequeña estructura que sirve como prueba del correcto funcionamiento.

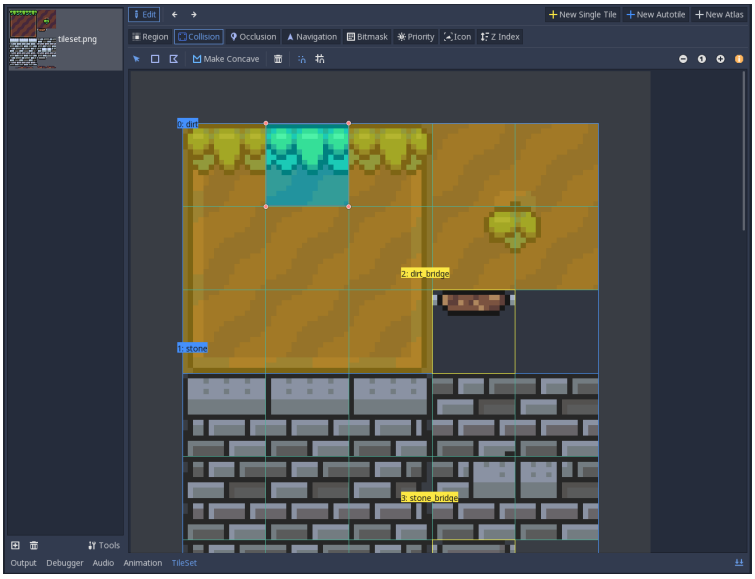


Figura 43: Definición de los tiles del tileset

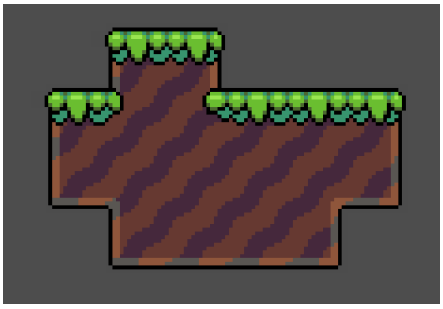


Figura 44: Plataforma de prueba

Después de comprobar que el TileMap funciona, es posible implementar el terreno del juego utilizando estos tiles. Además, se han añadido dos nodos Position2D en el mapa, uno en cada uno de los extremos. Este nodo no ofrece ninguna funcionalidad, pues sirve para marcar posiciones. Sin embargo, esto será útil más adelante cuando se instancien los jugadores, ya que estos puntos serán utilizados como spawns (posición en la que se ubica el jugador al comenzar la partida o al reaparecer después de morir).

6.3.1.2 Desarrollo del personaje base

Es momento de crear ahora los distintos personajes. Ya que todos ellos tienen unas funciones comunes, siendo sus diferencias la apariencia y los ataques, es posible utilizar herencia para desarrollar un personaje con toda la funcionalidad base (moverse, recibir daño, etc) y a continuación desarrollar las funciones individuales de cada uno de los diferentes personajes en una escena heredada propia.

Para realizar el personaje base se utilizará el nodo KinematicBody2D como base para la escena. Este nodo posee métodos específicos para el desarrollo de personajes controlados por el jugador, siendo el más importante de ellos «move_and_slide». Este método recibe como parámetro dos vectores de dos dimensiones: el primero es el que le indica la dirección en la que debe moverse, siendo el módulo de éste su velocidad; y el segundo indica en que dirección está «arriba» para el personaje. Éste segundo es importante para tener en cuenta

la diferencia entre una colisión con el suelo y la colisión con una pared, ya que viene dada en función del angulo que formen con dicho vector.

En el script de este nodo se utilizan dos métodos importantes para el desarrollo de un juego, los cuales vienen dados por Godot: «_ready» y «_process». El método «_ready» es llamado cuando el nodo es instanciado por primera vez, y es donde se realiza la configuración necesaria del personaje. Por otro lado, el método «_process» se llama en cada frame del juego, siendo generalmente y de forma predeterminada 60 veces por segundo. En este método es donde se deben comprobar los inputs del usuario para así actuar en consecuencia.

Otro aspecto importante es la forma en que se debe controlar a los personajes, puesto que cada personaje sólo debe poder recibir instrucciones por parte del jugador que lo controla. Por esto es que la solución sencilla, consistente en comprobar las teclas o botones que pulsa el jugador en cada frame mediante el método «_process» y actuar en función de las que haya pulsado, no es suficiente. Esto se debe a que todos los nodos de los jugadores existen a la vez en todas las máquinas que se encuentran en la partida, por lo que los inputs de un jugador afectarían a todos los personajes de la partida.

La solución a este problema consiste en usar la funcionalidad de «master» y «puppet» que se explicó anteriormente. Por defecto, es el servidor el que tiene autoridad sobre todos los nodos del juego. Sin embargo, se puede asignar a un nodo especifico (y con ello también a sus hijos) un «master» diferente al server. Si se hace esto, se pueden definir funciones a las que sólo pueda acceder quien tenga autoridad sobre el nodo. Por tanto, si en el momento de crear los personajes se asigna como «master» de cada uno al jugador que lo controla y se definen las funciones de control para que sólo puedan ser utilizadas por él, se limita el control de cada personaje a un único jugador.

Además, hay que tener en cuenta cómo se van a crear los jugadores. De esto se encarga el server, siendo éste el que almacena la información de todos los usuarios antes de iniciar la partida. Recordemos que, tal y como se explicó en el apartado del desarrollo del lobby, es el propio server el que detecta el momento en que la sala está llena y la partida puede comenzar, cambiando la escena a aquella del mapa. Es en ese momento cuando los personajes deben ser creados, asignados al jugador correspondiente, e instanciados en el mapa. Para ello se crea un método de inicialización en el personaje base. En él se proporcionan los datos del jugador que controla al personaje: su nombre y el equipo en el que participa. El nombre será mostrado en una etiqueta sobre el personaje, para que el resto de jugadores puedan distinguir quién controla a cada personaje. El equipo se utilizará para asignar las capas y máscaras del jugador según se definieron en la tabla 50. Tras la llamada a este método, el server instancia el jugador en el spawn correspondiente a su equipo, obteniendo la posición de éste mediante el nodo Position2D definido previamente.

Es en este momento cuando se llama al método «_ready». En él se realizan dos acciones: la inicialización de la interfaz gráfica y la de la cámara. La primera se realiza para todos los personajes, y es cuando se crea interfaz que se ubica sobre el personaje y que contiene la etiqueta que muestra el nombre del usuario y una barra de vida. Sin embargo, la cámara

que apunta a un personaje dado sólo debe estar disponible para el jugador que controla a dicho personaje. Para ello, y ya que anteriormente se ha definido quién es el master de cada jugador, se puede hacer una comprobación para saber si el jugador de la máquina actual tiene autoridad sobre cada personaje que se instancia y, en el momento en que la tenga sobre uno de ellos, se añade la cámara a dicho personaje.

Para el desarrollo de la interfaz gráfica de los jugadores se utiliza una escena personalizada. En ella se encuentra la etiqueta que muestra el nombre de usuario, y una barra de vida. Esta segunda está implementada mediante una modificación del nodo TextureProgress, que originalmente está ideado para desarrollar barras de carga. Sin embargo, modificando las texturas que utiliza y con un script personalizado, puede ser usada como barra que muestra dinámicamente la salud del personaje. Para ello, en primer lugar se debe inicializar con sus valores máximos y mínimos. Este paso se hace desde el método «_ready» del jugador base, ya que dispone de una constante que representa su vida máxima. Además, en el script de la interfaz se crea un método que responderá a una señal que emitirá el jugador cuando se dé un cambio en su salud. Este método recibirá la información de la salud actual y actualizará la barra para representar el porcentaje adecuado. En la imagen 45 se puede ver la interfaz del personaje con su barra de vida al 75 %.

Figura 45: Interfaz de los jugadores

Por otro lado, la cámara no necesita mucho desarrollo, pues Godot dispone del nodo Camera2D, que se utiliza para mantener la pantalla de un usuario enfocada en el nodo que la contenga. Por ello, con crearla sobre el personaje en la máquina que tenga autoridad sobre él, sería suficiente para que sea funcional. Sin embargo, se ha añadido una funcionalidad extra: poder mover la cámara por el mapa independientemente del personaje al que esté unida. Para ello se ha creado un script sobre la cámara que, en el método «_process», comprueba los inputs del usuario sobre las teclas asignadas para mover la cámara (las flechas del teclado). En función de éstas, puede añadir un offset entre el centro de la cámara y el personaje, con lo que permite moverla libremente para poder mirar el espacio alrededor.

Volviendo al personaje, su inicialización queda terminada. Por tanto es necesario implementar la funcionalidad que se debe ejecutar en todo momento: el movimiento y los ataques. Se debe tener en cuenta de nuevo que el usuario siempre debe depender de las ordenes del server, por lo que sus comandos para el movimiento del personaje deben ser enviados al server para que sean procesados en él y quedar a la espera de que el servidor mande la instrucción de mover el personaje en todas las máquinas de la red a la vez. Para realizar

esto, se utiliza un diccionario que almacena todos los inputs del usuario. Este diccionario es a continuación enviado al servidor, que llamará a un método que procesa ese diccionario y realiza el movimiento. A continuación, este mismo método es llamado remotamente desde el servidor con los inputs que el jugador original envió.

El movimiento, como se explicó anteriormente, se gestiona mediante vectores en dos dimensiones. Para definir cuál es el vector en cada frame se utilizan los inputs del usuario. La coordenada X del vector responde al movimiento horizontal, y se gestiona según las teclas definidas para el movimiento a izquierda y derecha. Si sólo una de ellas está pulsada, se asigna el valor de la velocidad de movimiento (definida en una constante) a la componente X, negativa en caso de ser la izquierda y positiva en caso de ser la derecha. En caso de que, además de lo anterior, la tecla de correr esté pulsada, se asigna el valor de la velocidad de carrera (definida en otra constante). Por último, en caso de que ninguna o ambas teclas de movimiento se encuentren pulsadas, el valor de la coordenada X será 0.

Por otro lado, la gestión del movimiento en el eje Y es más complicada. En un primer momento, en caso de que la tecla de salto se encuentre pulsada y si el personaje se encuentra en el suelo, a la coordenada Y del vector de movimiento se le asignará la velocidad de salto en valor negativo (puesto que para Godot, el eje Y crece hacia abajo y decrece hacia arriba). Además, se dispone de un doble salto, por lo que si el personaje se encuentra en el aire puede realizar un único salto adicional, asignando de nuevo el valor de la velocidad de salto al vector. Para poder saber si se ha realizado este doble salto existe una variable que actúa como flag, siendo positiva en caso de que se pueda realizar el doble salto y negativa cuando no. Esta variable se reinicia a positiva cuando el personaje toca el suelo.

Por último, se debe aplicar la gravedad al vector de movimiento antes de utilizarlo para realizar el propio movimiento. Para ello, en cada frame se almacena la velocidad vertical del personaje. Después de aplicar los inputs, en caso de que el jugador no haya realizado un salto, se asigna a la coordenada Y el valor que tuvo en el anterior frame con un pequeño incremento. De esta forma, la velocidad de caída aumenta en cada frame que el jugador se mantenga cayendo hasta que éste toque el suelo, en cuyo caso la velocidad sobre el eje Y vuelve a ser 0.

Tras realizar todas estas comprobaciones, es posible usar el método «move_and_slide» con el vector final obtenido para aplicar el movimiento necesario. Sin embargo, se ha realizado un añadido para optimizar el funcionamiento del movimiento y evitar situaciones en las que el jugador se encuentra en una posición diferente en su máquina y en el server. Esto se puede deber a que el protocolo que se utiliza por defecto en la NetworkedMultiplayerENet es UDP. Este protocolo es el más adecuado para aplicaciones como videojuegos, pues permite el envío rápido de pequeños paquetes de información. Sin embargo, no se realiza ninguna comprobación para asegurar que los paquetes enviados llegan al destino, con lo que en el caso de este proyecto puede que se creen desincronizaciones entre las posiciones del personaje si no se ejecuta el movimiento de un personaje durante uno o más frames. Para solucionar esto, después de realizar el movimiento, el servidor envía a su vez la posición final en la que termina

el personaje, pero en este caso utilizando un protocolo TCP. Con este protocolo, aunque más lento que UDP, es seguro que la información llegará a los clientes del server. Cuando llega, se fuerza la posición del personaje a la recibida. En una situación normal, la situación forzada debería coincidir con la que tiene tras el método del movimiento, por lo que el jugador no nota nada. Sin embargo, si se han perdido paquetes, de esta forma se asegura que el cliente siempre se encuentra sincronizado con el servidor. Esto además minimiza los problemas que puede provocar que la conexión de uno de los jugadores sea mala o se desconecte durante unos instantes.

La última comprobación que se debe hacer con estos inputs es la que sirve para los ataques de los jugadores. Estos, igual que el movimiento, se realiza mediante la notificación al servidor y consecuente llamada remota a los nodos. Para este prototipo se han creado los métodos que los diferentes personajes implementarán para sus ataques, aunque sin ninguna funcionalidad, puesto que esta será desarrollada en la creación de los personajes finales haciendo «override» a los métodos recién creados.

Por último, el personaje debe implementar los métodos que le permitan interactuar con otros jugadores y el entorno. El primero de ellos es aquel que se usa para recibir daño, en el que se pasa como parámetro el número de puntos de salud que se sustraen al personaje. Otro es el método para morir, que es llamado cuando la vida del personaje se reduce a cero, y tras un tiempo éste vuelve a aparecer en su spawn. Por último, se ha creado un método para recuperar salud, actuando si el personaje no ha recibido daño durante un periodo de tiempo y recuperando la salud de éste progresivamente.

Terminado el código interno del script del personaje, se desarrollará el apartado gráfico básico, compuesto principalmente por dos tipos de nodo: sprites y áreas de colisión.

Los sprites son las imágenes que formarán el cuerpo del personaje. Si bien es posible utilizar un único sprite para crearlo, se ha decidido usar diferentes «trozos» para que estos formen el cuerpo en conjunto. Esto se debe a que más adelante será necesario animar el personaje, y dividirlo en diferentes sprites simplifica la operación. En caso de utilizar un único sprite al completo, para cada «fotograma» de la animación es necesario crear un sprite diferente, lo que conlleva mucho trabajo. Sin embargo, si se utilizan diferentes sprites, es posible animar al personaje moviendo individualmente los sprites de cada parte de su cuerpo, obteniendo un resultado similar pero con una carga de trabajo mucho menor. Además, mediante este método es más sencillo obtener un resultado que realmente ofrezca sensación de movimiento en la animación.

Las áreas de colisión son las zonas en las que el personaje interactuará con su entorno. En el caso del personaje base, el único área necesaria es la de su propio cuerpo, mediante la cual se calculan las colisiones con el terreno y las interacciones con los ataques de otros personajes. Otras áreas serán utilizadas para la implementación de los ataques de los personajes, indicando la zona en que dicho ataque hace daño. Sin embargo, como se comentó anteriormente, en el personaje base sólo existen métodos vacíos para los ataques, por lo que no es necesario asignar ningún área más a él.

En la figura 46 se puede observar el resultado final. En ella se ve el área de colisión como un rectángulo de color azulado. A su lado, en la figura 47, se muestra al personaje con sus sprites individuales separados, para poder apreciar mejor la forma en que está construido. Conviene recordar que este personaje es la base de resto, por lo que este aspecto gráfico es temporal y únicamente servirá para comprobar un funcionamiento básico correcto.

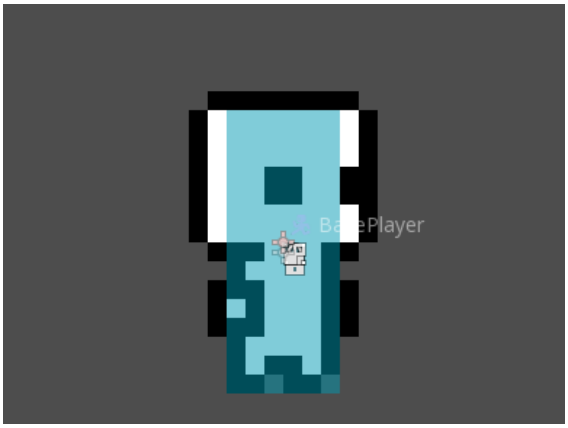


Figura 46: Personaje base

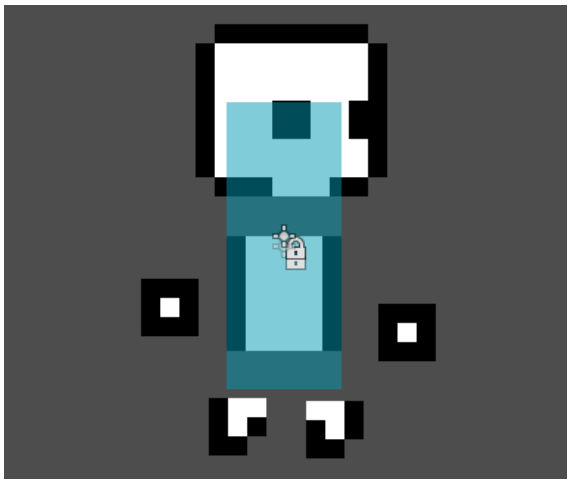


Figura 47: Personaje base desarticulado

6.3.1.3 Desarrollo de las estructuras

El último elemento necesario para que se pueda dar una partida completa son las estructuras que los jugadores deben destruir para ganar la partida. En Brawlchemy existen dos estructuras diferentes: las torres y el nexo.

En primer lugar se han creado las torres. El apartado gráfico de éstas es bastante simple, pues constan de pocos componentes: su sprite, el área de colisión y en la que reciben los ataques, y el área de detección de enemigos. Ésta última representa la zona en la que la torre podrá atacar a los enemigos que se encuentren en su interior. A diferencia de los personajes jugables, las torres se encuentran estáticas, por lo que no es necesario preocuparse de ninguna animación. Además de las torres, es necesario crear una escena que represente a los proyectiles con los que estas atacarán. Ésta se ha creado mediante un área que detectará la colisión con el jugador objetivo, así como un sprite personalizado.

Dentro del script de las torres se encuentra el método que ejecuta los disparos. Éste, en cada frame, comprueba si existe tanto un jugador aliado como un jugador enemigo dentro del área de detección de enemigos. En caso de que existan ambos, creará un proyectil al que asignará como objetivo el jugador enemigo que lleva más tiempo dentro del área. Además, se utiliza un temporizador que marcará el momento en que la torre puede volver a disparar, pues en caso contrario se encontraría disparando continuamente siempre que las condiciones

anteriores se cumplan. La condición de que ambos jugadores deben estar en esta área para que la torre ataque se utiliza para dar la posibilidad al jugador enemigo de acercarse a la propia torre para intentar destruirla siempre y cuando no exista un jugador enemigo en las inmediaciones. En caso de que la torre siempre pudiese disparar a los enemigos, sería demasiado complicado que estos la pudiesen destruir. Por último, se crea un método para poder dañar a la torre, siendo éste el que los ataques de los jugadores enemigos activarán, así como otro para destruir la torre cuando la salud de esta llegue a cero.

Por otro lado, el proyectil también dispone de un script personalizado para dotarlo de funcionalidad. Uno de sus métodos es el de inicialización, en el que se indica al mismo cuál es su equipo (para poder configurar correctamente la máscara de detección de jugadores) y cuál es su jugador objetivo. El otro método importante es el que gestiona el seguimiento del proyectil al jugador objetivo, ubicado dentro del método «_process» para que sea ejecutado en cada frame. En él, se se calcula el vector que une la posición del proyectil con la posición del objetivo. Este vector es normalizado (haciendo que su módulo sea 1 pero manteniendo su ángulo) y a continuación multiplicado por la velocidad base del proyectil. Tras esto, se mueve este proyectil a lo largo del vector calculado. Por último, se ha de detectar en el momento en que colisione con el personaje objetivo, por lo que se utiliza una señal que será emitida cuando el cuerpo de dicho objetivo se encuentre sobre el proyectil. En ese momento, se llamará a la función para dañar al personaje que se implementó en el personaje base, pasando como parámetro el daño base del proyectil. Tras esto, el proyectil desaparece.

En la figura 48 se muestra el resultado final de una torre.

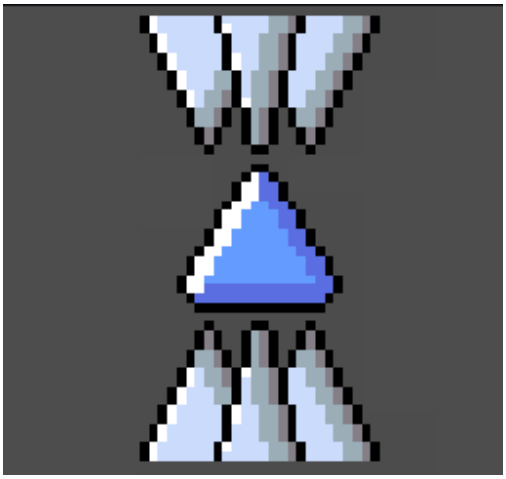


Figura 48: Estructura de una torre del equipo azul

La otra estructura que se debe crear es el nexo. Éste es similar a una torre, salvo que no dispone de la funcionalidad de disparar proyectiles. Por ello, su estructura es muy similar, difiriendo únicamente en que el nexo no dispone de un área de detección de enemigos y en que su sprite es diferente. Sin embargo, además de los métodos para dañarlo y destruirlo, hay que tener en cuenta que la destrucción del mismo es el objetivo último de la partida.

Por ello, el nexo dispone de una señal que es lanzada cuando éste es destruido, siendo esta la que más adelante podrá ser utilizada para la gestión del final de la partida. En la imagen 48 se puede ver el resultado del nexo.

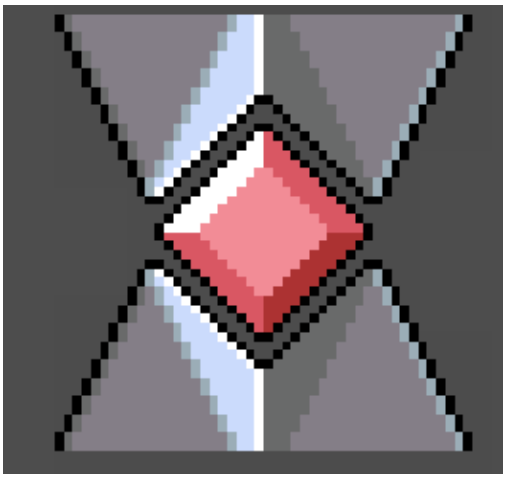


Figura 49: Estructura del nexo del equipo rojo

6.3.2. Pruebas

La realización de las pruebas de este prototipo se centrará en dos apartados: la comprobación de que las escenas realizadas ejecutan correctamente sus funciones cíclicas (como el control del personaje o la detección continua de enemigos por parte de la torre) son capaces de interactuar entre sí de forma correcta. Además, para simplificar, se dividirán las pruebas por escenas en caso de ser posible.

6.3.2.1 Pruebas del terreno

En la tabla 51 se muestran las pruebas realizadas sobre el terreno que forma el mapa de Brawlchemy.

Tabla 51: Resultado de las pruebas del terreno

Prueba	**Resultado esperado**	**Resultado obtenido**
Visualización correcta de los tiles	Los tiles encajan y se visualizan correctamente.	OK
Mapa cerrado	No existen espacios a través de los cuales un jugador pueda salir de los límites del mapa y/o caer al vacío.	OK

6.3.2.2 Pruebas de las torres

En la tabla 52 se muestran las pruebas realizadas sobre el funcionamiento de las torres.

Tabla 52: Resultado de las pruebas de las torres

Prueba	**Resultado esperado**	**Resultado obtenido**
Inicialización	La torre se muestra diferente dependiendo de su equipo.	OK
Disparos	La torre dispara únicamente cuando un aliado se encuentra cerca y en intervalos fijos de tiempo.	OK
Recibir daño	La torre puede ser dañada por jugadores del equipo enemigo.	OK
Ser destruida	Cuando la salud de la torre llega a 0, ésta es destruida.	OK

6.3.2.3 Pruebas de los nexos

En la tabla 53 se muestran las pruebas realizadas sobre el funcionamiento de los nexos.

Tabla 53: Resultado de las pruebas de los nexos

Prueba	**Resultado esperado**	**Resultado obtenido**
Inicialización	El nexo se muestra diferente dependiendo de su equipo.	OK
Recibir daño	El nexo puede ser dañado por jugadores del equipo enemigo.	OK
Ser destruido	Cuando la salud del nexo llega a 0, éste es destruído y emite una señal avisando de ello.	OK

6.3.2.4 Pruebas del personaje base

En la tabla 54 se muestran las pruebas realizadas sobre el funcionamiento del personaje base.

Tabla 54: Resultado de las pruebas del personaje base

Prueba	**Resultado esperado**	**Resultado obtenido**
Interfaz	En la interfaz del jugador se muestra su nombre y salud.	OK
Movimiento	El jugador se puede mover libremente.	OK
Correr	El jugador se mueve más rápido si pulsa la tecla de correr.	OK
Salto	El jugador puede realizar un salto y un doble salto correctamente.	ERROR 1
Ataque débil	El jugador llama al método de su ataque débil.	OK
Ataque fuerte	El jugador llama al método de su ataque fuerte.	OK
Ser dañado	El jugador puede ser dañado por enemigos y este daño se refleja en su barra de salud.	OK
Morir	Cuando su salud llega a 0, el jugador muere.	OK
Reaparecer	Tras morir, al cabo de un tiempo el jugador reaparece en su spawn.	OK

Resolución de errores:

- **ERROR 1:** Cuando el personaje cae por un borde del mapa (es decir, cuando éste se encuentra en el aire sin haber realizado previamente un salto) no se actualiza el flag que diferencia entre un salto normal y un salto en el aire (o doble salto). Por ello, en esta situación el personaje puede realizar dos saltos aéreos en vez de uno. La solución ha sido añadir una comprobación en todos los frames para que en caso de que el jugador se encuentre en el aire, su flag de salto se desactive y únicamente disponga de su doble salto.

6.3.2.5 Pruebas interacciones

En la tabla 55 se muestran las pruebas realizadas sobre las interacciones entre el terreno, los personajes y las estructuras.

Tabla 55: Resultado de las pruebas de las interacciones

Prueba	**Resultado esperado**	**Resultado obtenido**
Colisión con el terreno	Los jugadores no pueden atravesar los tiles del terreno del mapa.	OK
Colisión con las torres	Las torres sólo pueden ser atravesadas por jugadores de su mismo equipo.	OK
Colisión con el nexo	El nexo no puede ser atravesado por ningún jugador.	OK
Colisión entre jugadores	Los jugadores sólo colisionan con los del equipo enemigo.	OK
Seguimiento de los disparos	Los disparos se dirigen de forma continua hacia el jugador objetivo.	ERROR 2
Colisión de los disparos	Los disparos colisionan y dañan únicamente con el jugador objetivo de éstos.	OK

Resolución de errores:

- **ERROR 2:** Los disparos no parecen seguir al personaje objetivo. Esto se debe a un mal uso de las coordenadas de ambos en el método que calcula la dirección del disparo, pues en Godot un nodo tiene dos coordenadas diferentes: las globales, referentes a la escena; y las locales, referentes a su nodo padre. En este caso es necesario utilizar las coordenadas globales de ambos, pero se usaban las locales. El personaje tiene como

padre la propia escena, con lo que las coordenadas obtenidas mediante ambos métodos son las mismas. Sin embargo, el nodo padre del disparo es la torre que lo crea, por lo que sus coordenadas provocaban el fallo. Cambiando la forma de obtener dichas coordenadas soluciona el error.

6.4. Control del estado de la partida

Durante el desarrollo de este prototipo se implementará el control del estado de la partida, tanto el inicio como el final de ésta, así como el almacenamiento de los resultados en el servidor.

6.4.1. Desarrollo

Como se ha definido anteriormente, el estado de una partida de Brawlchemy sólo depende del estado de los nexos, puesto que es en el momento en que uno es destruido cuando la partida termina. Sin embargo, esto no es lo único que se debe monitorizar, ya que es necesario almacenar el número de veces que cada personaje muere y es matado por otros.

Para realizar esto, es necesaria una manera de monitorizar y almacenar las muertes de los jugadores. Anteriormente ya se ha visto la utilidad que ofrecen las señales en Godot, y en este caso serán utilizadas de nuevo. Es en el momento en que un personaje recibe daño cuando se calcula si éste ha muerto. En caso afirmativo, nos encontramos en un método en el que disponemos de la información de la persona que ha dañado al jugador, así como de la información del propio jugador, por lo que se ha modificado este método para que se emita una señal que transmitirá el identificador del jugador que ha matado y del que ha muerto.

Esta señal es utilizada en el script del mapa en la parte del server. Cuando se inicializa, se crea una variable que almacenará los datos de cada jugador relativos a la partida. Cada vez que se active la señal indicando que un jugador ha muerto, un método se encargará de aumentar las muertes y los asesinatos del jugador que ha muerto y del que ha matado respectivamente. Además, hay que considerar el caso en el que es una torre la que mata a un jugador, ya que las torres también pueden hacer daño a los jugadores. En este caso no sería necesario adjudicar esa muerte a ningún jugador. Por ello se ha creado otro método que diferencia entre estas dos situaciones para almacenar los datos de forma correcta.

Por otro lado, es necesario gestionar la finalización de la partida y, tras ello, el cierre automático del servidor. Cuando un nexo es destruido se emite una señal avisando de ello. En este momento, es necesario se den tres condiciones para que el servidor pueda ser cerrado: la partida tiene que ser actualizada y marcada como terminada, las participaciones de los jugadores deben ser añadidas, y los jugadores deben abandonar la propia partida, ya que si es el servidor el primero en cerrarse los jugadores serían expulsados de forma brusca y sin previo aviso. Para mantener una monitorización de cuáles de estas condiciones se han cumplido, se crea un flag para cada una de ellas.

La actualización de la partida en la base de datos se ha realizado mediante una petición HTTP simple. Ya que el server dispone del id de la partida, simplemente se debe enviar a la dirección correspondiente que se quiere actualizar el estado al correspondiente a una partida terminada. En el momento en que se recibe una respuesta positiva por parte del servidor, se

puede actualizar el flag correspondiente para reflejarlo.

Para añadir las participaciones se ha utilizado un método similar al que actualiza las participaciones. Sin embargo, en este caso debemos realizar varias peticiones, pues son varios jugadores los que participan en la partida. Para ello, en el flag se almacena el número de participaciones que se han añadido de forma correcta a la base de datos. Ya que se sabe que este número es igual al número de jugadores que participan en una partida, se puede saber el momento en que todas han sido realizadas.

Además, se debe permitir que todos los usuarios abandonen la partida sin ser forzados a salir. Por eso, el server esperará a que el número de jugadores conectados sea cero. Entonces el servidor actualiza el flag correspondiente para marcar que estos han abandonado la partida.

Por último, cuando los tres flags se encuentren activados, el server lo detectará y llamará al método que Godot proporciona para cerrar la instancia del juego.

Por otro lado, en la parte de los clientes es necesario ofrecer un aviso de que la partida ha terminado, indicando el ganador y dando además la opción de salir de ella. Para ello se ha creado una nueva escena nueva. Esta escena se ubica como hija de la cámara de cada usuario, ya que es necesario que el aviso aparezca en su pantalla sin importar la ubicación en la que se encuentre su cámara. En el jugador se ha creado un nuevo método, siendo este el que será llamado por parte del mapa cuando la partida termine. En él se pasa como argumento el equipo ganador, y es el propio jugador el que emite una señal avisando de ello. Esta señal se ha conectado a la cámara, siendo esta la que inicializa la escena y la muestra al usuario.

Para el desarrollo de la propia escena, se ha seguido un método similar al que se usa para los menús, basando la estructura en contenedores. En estos se encuentra una etiqueta indicando si se ha ganado o perdido la partida, un background dependiendo también del resultado, y un botón que servirá para abandonar la partida. En el momento de pulsar este botón, el juego se desconecta del server que alojaba la partida, elimina la información almacenada localmente referente a esta partida (nombre de jugadores, personajes, etc) y, tras ello, envía al usuario al menú principal.

En las imagenes 50 y 51 se puede ver la diferencia entre la pantalla de victoria y derrota.

Figura 50: Mensaje de victoria

Figura 51: Mensaje de derrota

6.4.2. Pruebas

Para la realización de las pruebas del control del estado de la partida ha sido necesario simular una situación de juego real. Para hacer esto más fácil se han modificado temporalmente ciertos parámetros de la partida, puesto que sólo se dispone de una persona para hacer las pruebas. Algunos de los cambios realizados son el número de jugadores necesarios para comenzar una partida (reducido de 6 a 2), el daño necesario para destruir las estructuras o la velocidad de los personajes, entre otros. Todos estos cambios agilizan la realización de las pruebas y minimizan los recursos necesarios.

En la tabla 56 se muestran las pruebas realizadas y su resultado.

Tabla 56: Resultado de las pruebas del control del estado de la partida

Prueba	**Resultado esperado**	**Resultado obtenido**
Actualización de la partida	La partida es actualizada correctamente en la base de datos.	OK
Creación de las participaciones	Las participaciones son creadas y enviadas correctamente a la base de datos.	OK
Notificación a los usuarios	La notificación aparece correctamente y muestra un mensaje correcto correspondiente a la victoria o derrota de cada usuario.	ERROR 1
Abandono de la partida	Los jugadores pueden abandonar la partida mediante el boton disponible en la notificación.	OK
Espera a las condiciones	El servidor espera que todas las condiciones se cumplan y reacciona a los cambios en éstas, quedando a la espera en caso contrario.	OK
Cierre del servidor	El servidor se cierra completamente cuando todas las condiciones se cumplen.	OK

Resolución de errores:

- **ERROR 1:** Cuando la partida termina, el mensaje que se muestra en inverso al esperado (mensaje de derrota en caso de victoria y viceversa). Esto se debe a que al ser destruidos, los nexos emiten una señal en la que comunican el equipo al que pertenecen. En pasos posteriores, esta información es tomada como el equipo ganador, siendo el equipo cuyo nexo es destruido el perdedor. Esto se ha podido solucionar de forma sencilla modificando la forma de comprobar el ganador teniendo en cuenta de forma correcta la información recibida.

6.5. Diseño final de gráficos

Durante el desarrollo de este prototipo se crearán e implementarán los gráficos finales del juego para personajes, entorno e interfaces de usuario.

6.5.1. Desarrollo

En este punto del proyecto, el apartado que necesita más implementaciones de gráficos son los jugadores, puesto que sólo se ha implementado la base para ellos. En un primer momento se ha creado una escena propia para cada uno de los 4 personajes: la maga, el guerrero, la cazadora y el boxeador. Estas escenas heredan de la del jugador base, por lo que disponen de los mismos nodos y métodos que este.

El primer paso que se ha realizado en cada uno de los personajes ha sido la asignación de unos sprites propios, acordes a la temática de cada uno. El resultado del cambio de sprites se puede ver en las imágenes figuras siguientes.

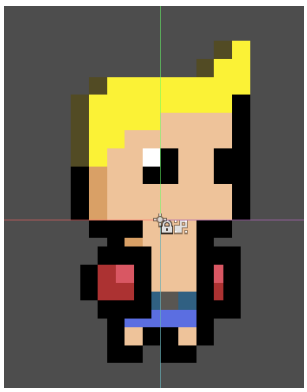


Figura 52: Boxeador.

Figura 53: Cazadora.

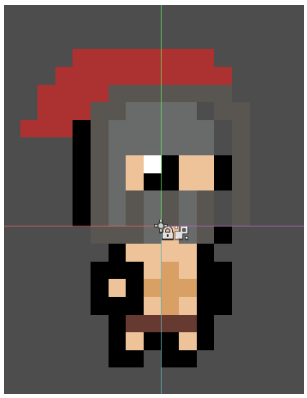


Figura 54: Guerrero.

Figura 55: Maga.

De la misma forma que el jugador base, estos sprites han sido divididos en varios sprites más pequeños. En el desarrollo del jugador base se explicó la utilidad de esto de cara a proporcionar animaciones a los jugadores, entrando ahora en juego. Haciendo uso de la herencia, es posible dotar al jugador base de animaciones básicas que todos los personajes utilicen, como las de moverse, correr o saltar, entre otras. Para implementar estas animaciones, se ha utilizado el nodo «AnimationPlayer». Este nodo permite crear distintas animaciones en base a otros elementos de la escena mediante cambios en su tamaño, posición, rotación, etc.

El proceso de la creación de una animación comienza definiendo un nombre y una duración. Tras esto, se deben definir las claves de la animación: pares de nodo y propiedad que se monitorizarán a lo largo del tiempo (por ejemplo el nodo de la cabeza y su rotación). Tras definir todas las claves necesarias, se debe definir cuál es el valor de éstas en cada momento dado. Sin embargo, no es necesario asignar un valor a cada frame de la duración de la animación, pues Godot dispone de varios tipos de transición entre valores de las claves, pero en el proyecto se utilizan principalmente dos: linear y más cercano. El primero de estos modos comprueba los valores que se ha dado a cada clave a lo largo de la duración, y entre éstas automáticamente asigna valores de forma que se produzca una transición uniforme entre los dos valores. Por otro lado, la transición al más cercano asigna a cada punto de la animación el valor más cercano entre los definidos por el desarrollador, por lo que entre cada par de valores se produce una transición abrupta.

Lo siguiente que se ha realizado ha sido la implementación de los ataques de los personajes. Estos ataques se presentan de dos formas: aquellos que utilizan armas o partes del cuerpo del personaje, y los que necesitan de la creación de proyectiles. En ambos casos se utilizará una animación personalizada para los ataques de cada personaje mediante el uso de un nuevo nodo «AnimationPlayer». Sin embargo, la implementación del propio ataque depende del tipo de ataque.

Para implementar aquellos ataques que utilizan armas o partes del cuerpo del personaje es posible utilizar el propio nodo que dota de animación al mismo. En primer lugar, y en caso de ser necesario, se ha añadido un nuevo sprite que represente el arma a utilizar. Tras esto, se ha creado un área de colisión en la zona que hará daño a los enemigos, y mediante un script se consigue que este área dañe a los enemigos que entren en ella. Sin embargo, no se desea que este área haga daño a los enemigos de forma continuada, sino sólo cuando el jugador se encuentre atacando. Es para esto para lo que es útil usar el nodo «AnimationPlayer», pues mediante él se puede activar y desactivar el área de colisión, además de dotarla de movimiento. La lista de los ataques que utilizan esta implementación se puede ver a continuación:

- Los ataques del guerrero utilizan una espada formada por un sprite y un área de detección de colisiones. Esta se mueve en forma de abanico alrededor del personaje para la realización de ambos ataques, siendo la principal diferencia entre ellos la velocidad a la que se realizan.

- Ambos ataques del boxeador se realizan mediante sus puños, teniendo cada uno de ellos un área de detección de colisiones. Estos se separan del cuerpo y vuelven a él de formas diferentes en su ataque débil y su ataque fuerte.
- Uno de los ataques de la maga consiste en un golpe con una vara, provocando una pequeña explosión. Tanto la vara como la explosión están formadas por un sprite propio, siendo la explosión la que dispone del área de detección.

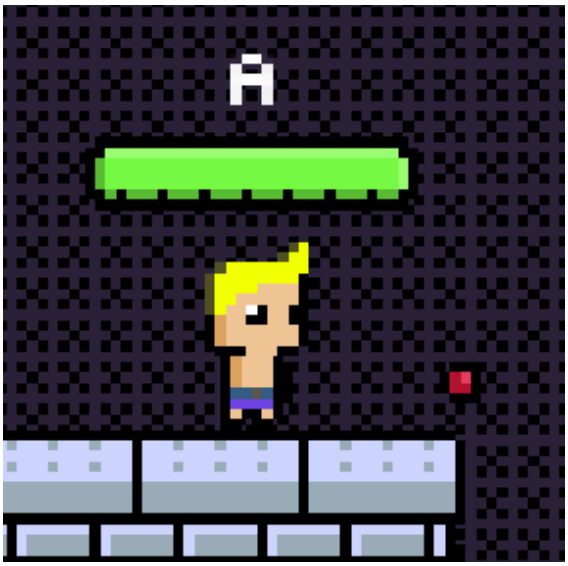


Figura 56: Ataque débil del boxeador.

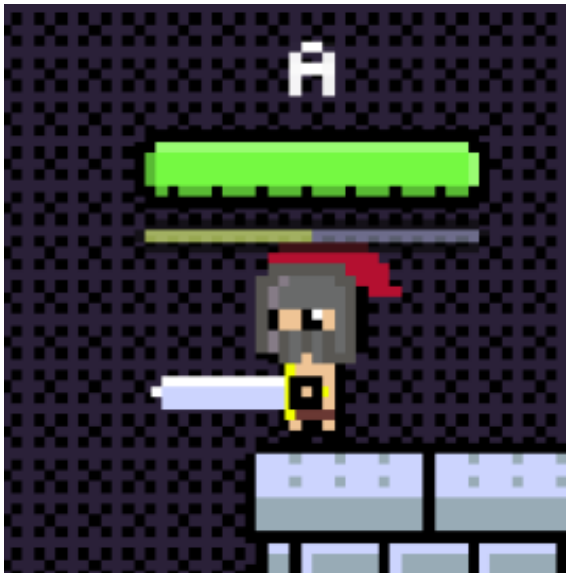


Figura 57: Ataque fuerte del guerrero.

Para el desarrollo de los ataques que utilizan proyectiles o similares, es necesario utilizar un método diferente. En un primer momento el desarrollo de la animación del ataque se realiza de la misma forma. Sin embargo, durante esta animación es necesario instanciar el proyectil que se va a utilizar, pero esto no se puede realizar mediante el nodo «AnimationPlayer» al no poder realizarse mediante una propiedad, ya que es necesario un método en un script. Para solventar esto existen tres posibles soluciones: instanciar la escena del proyectil antes de comenzar la animación, instanciarla después de que termine, o dividir la animación en dos e instanciarla entre medias. Dado el tipo las animaciones que se utilizan en este proyecto, las dos primeras opciones son las que se han sido utilizadas, pues la tercera es más apta para animaciones con un gran nivel de detalle y conlleva un mayor grado de complejidad.

Además de la animación en sí, es necesario desarrollar las escenas de los proyectiles. Todos ellos están compuestos por un área de detección de colisiones y un sprite. En esta escena ha sido necesario utilizar un script en el que se incluye un método que inicializa el proyectil (equipo al que pertenece, daño que realiza, etc), otro que lo mueve de la forma adecuada para dicho proyectil, y un último que reacciona a las colisiones con enemigos y los

daña. Además de esto, y dependiendo del proyectil, algunos incluyen otros nodos específicos que utilizan para implementar su funcionalidad. A continuación se muestra una lista con los ataques que utilizan proyectiles y sus características principales:

- El ataque débil de la cazadora consiste en un disparo de una flecha con su arco. Esta flecha se ha programado para moverse en linea recta hacia la dirección en la que mira el personaje y se destruye tras hacer daño a un jugador enemigo al impactar con él. Además, dispone de un temporizador que destruye la flecha al cabo de un tiempo si ésta no ha impactado contra nadie, para así evitar que recorra una distancia infinita.

- El ataque fuerte de la cazadora consiste en un bumerán. Este actúa de forma similar a la flecha en cuando a que es lanzado en la dirección a la que mira el personaje. Sin embargo, el bumerán no se destruye al colisionar con jugadores. También dispone de un temporizador que, a diferencia de la flecha, no lo destruye al terminar, sino que cambia la trayectoria del mismo para que vuelva hacia el personaje que lo ha utilizado. Es al llegar de vuelta al personaje que lo usa cuando el bumerán desaparece.

- El ataque fuerte de la maga consiste en una bola de fuego. Este ataque está compuesto por dos partes: en primer lugar, la bola de fuego traza un pequeño arco en la dirección en la que mira el personaje; a continuación, tras colisionar con un jugador enemigo o con el terreno, la bola de fuego crea una pequeña explosión que se mantiene en el mapa por unos segundos. Tanto la primera como la segunda parte del ataque hacen daño a los jugadores con los que colisionan.

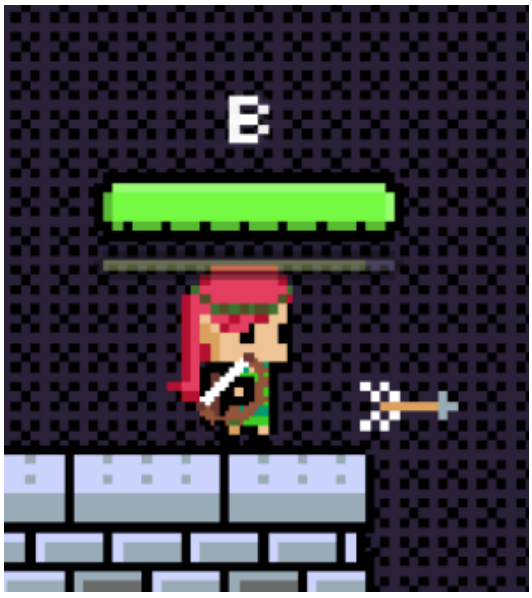


Figura 58: Ataque débil de la cazadora.

Figura 59: Ataque fuerte de la maga.

Para terminar con los personajes, se ha añadido una nueva interfaz de usuario donde se puede ver la salud del jugador. Este añadido viene dado porque en las pruebas de anteriores prototipos, al mover la cámara y hacer que el jugador que se controla saliese de los límites de ésta, no se puede saber la información del personaje. Para solucionarlo se ha asignado a la cámara una nueva interfaz de usuario que siempre va unida a ella, apareciendo en la parte inferior de la pantalla. El funcionamiento de esta interfaz es el mismo que el de la interfaz que se encuentra sobre el personaje, reaccionando a la señal que éste emite cuando su salud cambia.

Otro aspecto que también ha sido actualizado son los gráficos de los nexos y de las torres. Mediante el uso del nodo «AnimatedSprite» y con una serie de modificaciones del sprite que se utilizó anteriormente, se han animado tanto las torres como los nexos. Este nodo funciona de forma similar al nodo Sprite, con la excepción de que admite que se le asignen multiples imágenes y, mediante un pequeño script, se puede hacer que estas imágenes se muestren de forma consecutiva y cíclica una a una. De esta forma, al utilizar imágenes similares, se puede lograr un efecto de movimiento. En la figura 60 se muestra la utilización de este conjunto de sprites para formar la animación.

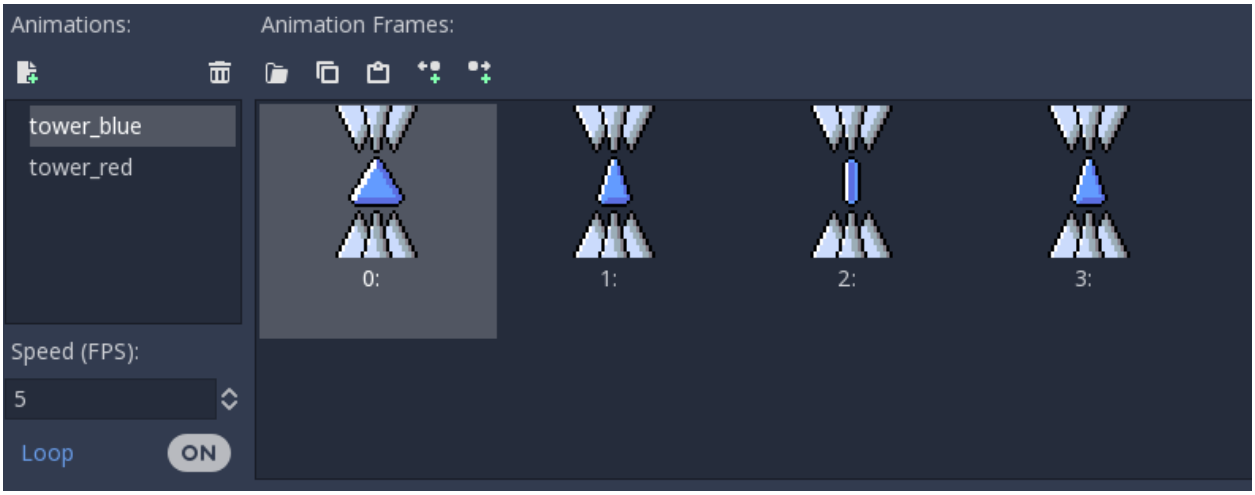


Figura 60: Frames utilizados para la animación de la torre.

Siguiendo con las torres, en pruebas anteriores se había visto que de cara a los jugadores es difícil saber cuándo una torre los va a atacar y cuando no, pues cabe la posibilidad de que un jugador enemigo se encuentre en las inmediaciones de la torre que se está atacando y, al estar lejos, no aparezca en la cámara. Para solucionar este problema se ha añadido también un efecto de partículas a la torres. Este efecto sólo es visible cuando la torre esté preparada para atacar, por lo que no es necesario ver a un jugador enemigo, bastando entonces con poder ver si la torre está emitiendo partículas. El resultado de este cambio se puede ver en la imagen 61.

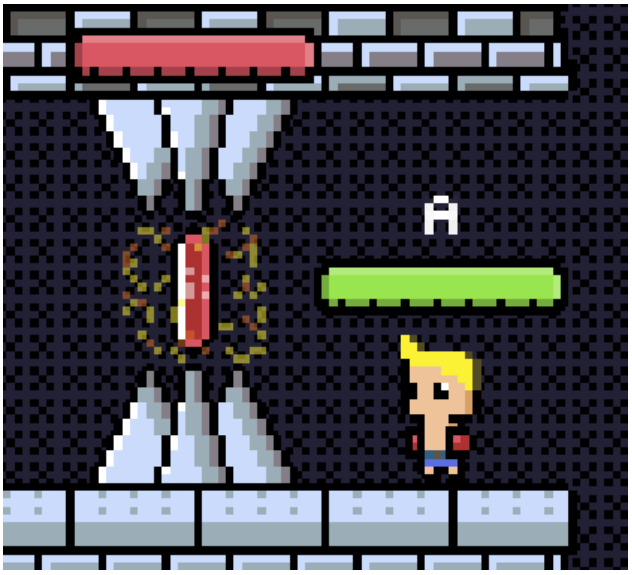


Figura 61: Efecto de partículas de la torre.

Además, también se detectó que en caso de haber varios jugadores cerca de la torre, en caso de que ésta dispare a uno de ellos puede ser difícil saber quién es el objetivo del proyectil. Para solucionarlo, se ha añadido al proyectil un método que dibuja una línea entre su objetivo y él, por lo que siempre es posible saber a quién va a atacar éste. El resultado de este añadido se puede ver en la figura 62.

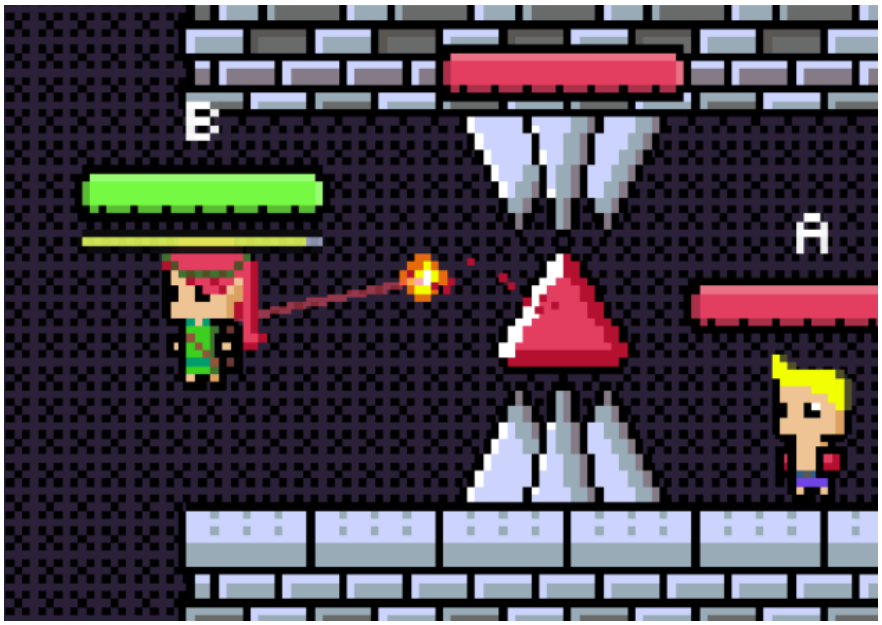


Figura 62: Efecto de apuntado del proyectil de la torre.

6.5.2. Pruebas

En la tabla 57 se muestran las pruebas realizadas y su resultado.

Tabla 57: Resultado de las pruebas del control del estado de la partida

Prueba	**Resultado esperado**	**Resultado obtenido**
Animaciones básicas de los personajes	Las animaciones de los personajes se ejecutan de manera fluida y de forma acorde a la acción que se esté ejecutando.	ERROR 1
Ataques del guerrero	Los ataques del guerrero funcionan de forma correcta, tanto gráfica como internamente.	OK
Ataques de la cazadora	Los ataques de la cazadora funcionan de forma correcta, tanto gráfica como internamente.	OK
Ataques de la maga	Los ataques de la maga funcionan de forma correcta, tanto gráfica como internamente.	OK
Ataques del boxeador	Los ataques del boxeador funcionan de forma correcta, tanto gráfica como internamente.	OK
Interfaz de la cámara	La interfaz de usuario de la cámara se muestra y actualiza de forma correcta reflejando la salud actual del personaje.	ERROR 2
Animación de torres y nexos	Las animaciones de las torres y los nexos se ejecutan de forma continua y fluida.	OK
Partículas de las torres	La torre emite un efecto de partículas cuando un jugador aliado se encuentra cerca.	OK
Apuntado de los proyectiles	Los proyectiles de las torres se conectan al jugador objetivo mediante una línea.	OK

Resolución de errores:

- **ERROR 1:** Cuando un personaje muere y se muestra la animación correspondiente, al reaparecer las animaciones quedan bloqueadas, no mostrándose ninguna desde ese momento. Esto se debe a que la animación que se ejecuta al morir no llega a finalizar correctamente tras ejecutarse debido a que el jugador debe desaparecer y aparecer en su spawn correspondiente. Al no haber terminado correctamente, las siguientes animaciones no pueden iniciarse y por tanto el sprite queda en un estado de bloqueo. Para solucionar este fallo, al momento de hacer reaparecer al personaje se fuerza la finalización de la animación, permitiendo que tras aparecer de nuevo se puedan ejecutar nuevas animaciones de forma correcta.

- **ERROR 2:** Cuando la cámara se mueve, la interfaz unida a ella tiene un retardo en el movimiento, cuando debería mantenerse de forma constante en la misma posición. Tras investigar el problema, se ha encontrado el origen del mismo: la ubicación de nodos como hijos de la cámara no asegura que estos se vayan a mostrar en pantalla, pues la función de ésta es simplemente actuar como una lente que captura el espacio que se debe mostrar en la pantalla del jugador y no debería tener nodos hijos que necesiten ser dibujados en ella. Sin embargo existe un nodo cuya utilidad es precisamente dibujar elementos en la pantalla sin importar su posición, llamado «ViewPort». Este nodo actúa como una última capa de procesamiento de la imagen, recibiendo como entrada la imagen a mostrar, añadiendo los elementos necesarios, y a continuación enviando como salida el resultado final a la pantalla del usuario. Por tanto, toda la funcionalidad que se encontraba en la cámara ha sido trasladada a uno de estos nodos, con lo que el error ha quedado arreglado.

7. Verificación final

Si bien ya se han definido pruebas unitarias para cada uno de los prototipos y se ha comprobado que Brawlchemy funciona correctamente, se puede definir una prueba que simule las condiciones reales de ejecución del juego: una prueba pública.

Mediante una prueba pública se puede comprobar que tanto el servidor como el juego soportan unas condiciones de uso prácticamente iguales a las que tendría en su ambiente de ejecución objetivo, siendo este el uso por parte de una gran cantidad de usuarios al mismo tiempo. Sin embargo, esta prueba tiene la ventaja añadida de poder estar monitorizando en tiempo real una gran variedad de parámetros: la carga del procesador del servidor, la velocidad de respuesta del mismo en función de la ubicación del usuario o del número de usuarios conectados simultáneamente, etc.

Para esto, sin embargo, es necesaria la participación de la máxima cantidad de personas posibles. Por ello, aparte de contactar con personas cercanas que pudiesen participar en la prueba pública, se decidió anunciar la necesidad de «testers» en la página web Reddit [14]. En ella existen diversas comunidades centradas en temas específicos, entre las que se encuentran la comunidad de Godot y la de desarrollo de videojuegos. Fue en estas dos comunidades donde se anunció esta prueba, siendo esta muy apoyada en la comunidad de Godot. Para poder llevar a cabo una comunicación básica con todos los participantes se utilizó Discord, una aplicación que permite la creación de diferentes salas de voz con las que comunicarse entre usuarios, mediante la cual se dieron las instrucciones básicas sobre la utilización del juego, la instalación y la forma de acceder a las partidas, así como para responder cualquier tipo de duda que los participantes pudiesen tener.

En el momento de la prueba, se logró reunir a casi 60 personas simultáneamente, una cantidad suficiente para su realización. Entre estas la gran mayoría utilizaron Windows como sistema operativo, habiendo sólo 7 personas utilizando macOS y 8 personas utilizando diferentes distribuciones de GNU/Linux. En un primer momento la prueba se realizó de forma incremental, creando partidas lentamente y comprobando como afectaba cada una de ellas al rendimiento del juego y del servidor durante algo más de 30 minutos.

Los resultados de esta prueba han sido gratamente satisfactorios. Por un lado, el funcionamiento del servidor REST no ha dado ningún problema y se ha comportado de forma correcta a lo largo de toda la prueba, incluso cuando éste era usado por todos los voluntarios de forma simultánea.

Por otro lado, en cuanto a las instancias server del juego, si bien es cierto que el rendimiento del estos no era el mismo que en las pruebas unitarias, éste era perfectamente utilizable y no suponía un gran problema de cara al usuario. Además, gracias a la monitorización que se realizó, quedó claro que el principal limitante era la capacidad de procesamiento del servidor. Si bien esta en ningún momento llegó a su máximo posible, pues se mantuvo alrededor del 70-80 % durante el tiempo en que todos los jugadores se encontraban a la vez en partida.

Por último, en lo referente al funcionamiento del propio juego de cara a los jugadores, algunos usuarios reportaron pequeños errores gráficos en las animaciones de personajes a la hora de atacar y recibir daño. Si bien parte de estos errores es probable que se diesen debido a las variaciones de latencia entre los clientes y el servidor, otros comentaron la ocurrencia de un glitch gráfico en el que en ciertos frames las texturas de algunos elementos de la pantalla aparecían «estiradas». Tras una investigación en internet, parece que este error es causado por el propio motor gráfico y será arreglado en la siguiente versión del mismo, por lo que no ha sido necesario realizar ningún cambio en la estructura del juego.

Con estos resultados, y en vista a los pocos problemas encontrados en una prueba de funcionamiento tan intensiva como la realizada, se puede decir que el proyecto Brawlchemy está completamente terminado.

8. Conclusiones y trabajo futuro

En esta sección se analizará de forma crítica todo el trabajo realizado: desde los resultados finales obtenidos en el proyecto, pasando por los problemas en el desarrollo y hasta la opinión personal referente al propio proyecto.

8.1. Cumplimiento de los objetivos

Al comenzar el proyecto se propusieron una serie de objetivos a cumplir para considerar el proyecto terminado con éxito. A continuación se listarán estos objetivos y se comprobará si estos han sido satisfactoriamente cumplidos y por qué.

- *Desarrollo de la base para un juego funcional y con todas las características básicas de un MOBA, así como con los cambios propuestos para que sea diferente al resto de juegos disponibles.*

  Brawlchemy cumple todos los requisitos que un videojuego del género MOBA debe cumplir: partidas por equipos, controlando a diferentes personajes con habilidades propias y con el objetivo de capturar una base enemiga. El hecho de haberlo realizado en 2D y simplificando el juego para hacerlo más accesible a un público nuevo hace que sea, con creces, diferente al resto de juegos disponibles, por lo que este objetivo se considera cumplido.

- *Implementación total del modo multijugador, esencial para un videojuego MOBA. Este modo debe ser totalmente transparente de cara al usuario, de modo que no necesite ningún tipo de configuración especial por su parte para poder utilizarlo.* Este objetivo está cumplido, pues Brawlchemy se puede jugar de forma multijugador sin ningún tipo de problema para el usuario. La única tarea que éste tiene que realizar es un registro en el sistema, con lo que cualquier persona puede utilizarlo con instalar el juego en su ordenador.

- *Creación de un juego multiplataforma. Es imprescindible que el juego pueda ejecutarse sin problemas en los principales sistemas operativos (Windows, Linux y Mac).* Aunque este objetivo no se pudo comprobar de primera mano en las pruebas realizadas tras cada prototipo, en la prueba pública realizada se pudo probar el funcionamiento correcto de Brawlchemy en los tres principales sistemas operativos, con lo que el objetivo queda cumplido.

- *Profundización y aprendizaje en las técnicas necesarias para el desarrollo de videojuegos de una forma profesional, por razones tanto académicas como por mera satisfacción personal de ver los resultados de un trabajo bien hecho.* Si bien este objetivo es subjetivo, personalmente considero que esta cumplido de sobra, puesto que ha sido necesario

aprender a utilizar una gran cantidad de tecnologías, así como entender el funcionamiento interno de estas. Esto ha sido especialmente importante para el desarrollo del apartado multijugador, ya que antes de entender el funcionamiento de Godot es necesario entender el funcionamiento de los apartados de bajo nivel, como son los distintos protocolos de transmisión de datos, comprender el funcionamiento de puertos y firewalls, entre otros. Esto, sin duda, ha aumentado y reforzado el conocimiento del que dispongo acerca del desarrollo de videojuegos a un nivel profesional.

Por otro lado, el resultado final es increíblemente satisfactorio, pues es el resultado de una gran cantidad de horas de trabajo invertidas en él y se asemeja a la idea que se tenía en mente desde un primer momento. Este proyecto es, con diferencia, el proyecto del que siento más orgullo entre todos los que he desarrollado a lo largo de la titulación.

8.2. Cumplimiento de la planificación

Al comienzo del proyecto, cuando se definió la Estructura de Descomposición del Trabajo (apartado 2.2.2) se asignó a cada tarea una duración estimada. Ahora, tras haber finalizado el proyecto, es posible realizar una retrospección para analizar en qué medida se cumplieron estas tareas, así como el efecto que esto tiene en el proyecto.

En la tabla 58 se muestra, de forma desglosada, las horas estimadas junto a las horas reales que han sido necesarias para desarrollar cada una de las tareas del proyecto. Cabe destacar que la Defensa del TFG se desarrollará de forma posterior a esta memoria, por lo que se seguirá utilizando su duración estimada para realizar los cálculos necesarios. Todas las duraciones afectadas por esto se han marcado con un asterisco (*).

Tabla 58: Duración final del proyecto

Tarea	**Duración prevista**	**Duración real**
Documentación	**90**	**97***
Elaboración del DOP	30	22
Elaboración de la Memoria del TFG	40	55
Elaboración de la Defensa del TFG	20	20*
Gestión	**26**	**19**
Reunión inicial	1	1
Reuniones con el director	5	4
Definición de los objetivos del proyecto	2	2
Definición de las tareas del proyecto	6	2
Desarrollo de la planificación temporal y económica	4	3
Gestión de riesgos	5	3

Definición de las herramientas a utilizar	3	4
Aprendizaje	**43**	**52**
Instalación de las herramientas necesarias	3	2
Aprendizaje sobre la arquitectura elegida	25	30
Aprendizaje sobre el motor elegido	15	20
Captura de Requisitos	**19**	**23**
Definición de funcionalidades	3	5
Definición de casos de uso	15	15
Definición del modelo de dominio	1	3
Análisis y diseño	**37**	**29**
Diagrama relacional de la base de datos	2	2
Diagrama de clases	15	12
Diagramas de secuencia	20	15
Prototipos	**100**	**115**
Base de datos y servidor	15	36
Menús, login y registro	15	25
Creación de los personajes jugables	25	20
Control del estado de la partida	25	12
Diseño final de gráficos	20	22
TOTAL	**315**	**335***

Si bien la duración final del proyecto parece ser similar a la esperada, la varianza en las tareas individuales ha sido bastante grande. La duración de la mayoría de ellas se ha visto incrementada ligeramente. Sin embargo, de forma opuesta, las tareas que han visto reducida su duración lo han hecho en gran medida, por lo que el resultado final parece compensarse.

A continuación, de entre estas tareas se han seleccionado aquellas más significativas en relación a las modificaciones en su duración:

- **Definición de las tareas del proyecto:** Si bien en un primer momento se consideró que el tiempo necesario para realizar esta tarea era relativamente alto debido a la gran cantidad de tareas a definir y explicar, al final resultó una tarea bastante sencilla y rápida, pues una vez creada una plantilla que se utiliza para cada tarea es posible acelerar mucho el proceso de su definición.
- **Base de datos y servidor:** Esta ha sido, con gran diferencia, la tarea que ha necesitado una mayor inversión de tiempo, pues la implementación del servidor se realizó con

un lenguaje (NodeJS) que no se había utilizado ampliamente. Además, la definición de pruebas unitarias mediante Postman, a pesar de haber sido de gran utilidad, también requirió de un gran tiempo de desarrollo. El principal motivo para no haber hecho un buen cálculo ha sido el no considerar esta herramienta como una nueva, pues al haber incluido previamente una tarea específica para el aprendizaje de la misma se esperaba una velocidad de desarrollo similar a aquella que se habría tenido con una herramienta más familiar, cuando esto no ha sido así.

- **Control del estado de la partida:** En un primer momento se consideró este prototipo como una parte de especial importancia en el proyecto, pues al fin y al cabo la gestión de la partida es lo que hace que el propio juego sea jugable y tenga final. Sin embargo, ya que en el momento de desarrollar este prototipo se dispone de todo el «esqueleto» del juego ya construido, la gestión de la partida se puede implementar de una forma más rápida de lo esperado.

8.3. Aparición de riesgos

Anteriormente, en el apartado referente a la Gestión de Riesgos (2.5), se definieron los posibles riesgos previstos para este proyecto. Durante el desarrollo de Brawlchemy se han dado tres de los riesgos definidos anteriormente.

En primer lugar, ha habido periodos de tiempo en los que la productividad ha sido muy baja. Esto se ha debido principalmente a que la carga de trabajo a lo largo del curso ha sido muy desigual, pues el proyecto se comenzó a finales del primer cuatrimestre y la planificación temporal se hizo en función de la carga de trabajo que se tuvo en ese momento. Sin embargo, esta carga aumentó considerablemente durante el segundo cuatrimestre, habiendo largos periodos de tiempo en los que el avance del proyecto se vio parado prácticamente por completo.

Por otro lado, durante la implementación del modo multijugador en Brawlchemy hubo diversos momentos de «bloqueo» que retrasaron en parte el desarrollo de este apartado. Estos bloqueos se debían principalmente a las pocas referencias que se encontraban respecto al desarrollo de juegos multijugador en Godot. Sin embargo, la gran variedad de apartados que podían ser desarrollados de una forma más o menos paralela permitía que, en caso de estar en medio de un gran bloqueo, fuese sencillo realizar una tarea menos intensa o más fácil, por lo que fueron pocos los momentos en los que la productividad se vio reducida a cero.

En consecuencia a los dos riesgos anteriormente mencionados, y como se refleja en el anterior apartado, la planificación temporal no ha sido correcta y ha tenido que ser modificada, siendo este otro de los riesgos previstos anteriormente. Sin embargo, al disponer de tiempo suficiente para la realización del proyecto, sólo ha sido necesario retrasar las tareas pendientes de forma proporcional al tiempo que se han visto retrasadas.

8.4. Mejoras y trabajo futuro

Si bien el resultado de este proyecto es un juego muy completo y se han cumplido con creces los objetivos del mismo, existen diferentes mejoras que podrían ser implementadas en un futuro. Algunas de estas mejoras se explican a continuación:

- **Música y efectos de sonido:** Uno de los principales objetivos personales que se planteaba con la realización de este proyecto era construirlo desde cero sin utilizar ningún recurso externo en el juego. Si bien esto ha sido posible con los gráficos y el código detrás de toda la implementación, en la actualidad no dispongo de los conocimientos necesarios como para aventurarme en desarrollar música o efectos de sonido propios. Durante el desarrollo del Documento de Objetivos del Proyecto se planteó considerar la música como una parte más del propio proyecto, pero en última instancia fue descartada por la gran carga de trabajo que conllevaría.

- **Mejora del servidor:** Como se pudo ver durante la prueba que se realizó con los voluntarios que se presentaron a ella, la máquina virtual que se usa actualmente como servidor no es suficiente como para mantener un juego multijugador como Brawlchemy. Al fin y al cabo la posibilidad del uso de una máquina más potente se reduce a un gran gasto de dinero en ella, por lo que se debería obtener unos beneficios suficientes con el propio juego como para sustentar los gastos que conllevaría este cambio.

- **Introducción de compras dentro del juego:** En la sección del Modelo de Negocio se explicó que el mejor modelo en un juego de las características de Brawlchemy es aquel que se basa en las microtransacciones, permitiendo a los usuarios realizar compras dentro del juego que les ofrezcan pequeños cambios o mejoras gráficas a sus personajes. Si bien esto no ha sido implementado en el proyecto, sería la forma de poder mantenerlo en un futuro de cara a una salida de Brawlchemy al mercado.

- **Diferentes lenguajes:** Actualmente el juego está completamente desarrollado en inglés, por lo que la posibilidad de trasladar el juego a diferentes idiomas podría aumentar la cantidad potencial de usuarios de la aplicación, pues a pesar de que el inglés es conocido y entendido por mucha gente, no todo el mundo dispone de una fluidez suficiente como para entender todos los posibles aspectos de la aplicación. Uno de los grupos más grandes que se podría beneficiar de esta mejora serían, por ejemplo, los niños pequeños cuyo idioma materno no sea el inglés.

- **Aplicación móvil:** El juego desarrollado dispone de unos controles suficientemente simples como para que la posibilidad de usarlo desde un smartphone sea totalmente factible, aunque llevaría una gran cantidad de trabajo para modificar todos los aspectos del mismo que nunca fueron pensados o diseñados para interactuar con una pantalla táctil.

- **Actualizaciones continuadas:** En cualquier juego MOBA como Brawlchemy es necesario ofrecer de forma periódica actualizaciones a los usuarios. Mediante estas actualizaciones se pueden arreglar errores del juego, pero sobre todo se añade nuevo contenido o se modifica el existente. Ya que este género se espera que los jugadores se mantengan jugando al juego de forma casi indefinida, es necesario ofrecer a los usuarios nuevo contenido y modificar el existente para que el juego no se estanque en el mismo estado y, con ello, los jugadores comiencen a abandonarlo.

8.5. Disponibilidad del proyecto

Como se comentó al comienzo de la memoria, Godot es un motor muy nuevo y que a pesar de tener muy buena documentación y una gran cantidad de demos, dispone de muy pocos proyectos que sirvan como ejemplo para funciones avanzadas, como es el desarrollo de juegos multijugador. Es por esto que, tras la realización y exposición de este trabajo, éste será publicado de forma gratuita en la librería de proyectos libres de Godot. En ella es donde se pueden encontrar multitud de proyectos centrados en el desarrollo de pequeñas funciones, centrados sobre todo en servir como tutoriales de la función en la que se centran. Además, al publicarlo de forma abierta se permite a otros usuarios aportar novedades al proyecto y mejorarlo en todos los aspectos, por lo que esto se ha considerado la mejor opción.

8.6. Reflexión personal

Con el proyecto ya terminado, además de todo el análisis objetivo en relación a los resultados del proyecto, es el momento de realizar una reflexión más subjetiva basada en la opinión personal del proyecto, tanto de los resultados como del propio desarrollo.

La primera sensación que surge al terminar un proyecto de esta envergadura es una gran satisfacción personal, pues después de tanto tiempo y trabajo se ven todos los objetivos cumplidos y eso, obviamente, hace que se sienta que todo el esfuerzo realizado ha merecido la pena. El resultado final del juego es muy similar a lo que se imaginaba al comenzar el proyecto, a pesar de que durante ciertos momentos y bloqueos pensase que fuese a ser necesario modificar algún objetivo por no ver soluciones a ciertos problemas. Es por este motivo que, a pesar de estas dificultades, es especialmente gratificante poder ver que todos los objetivos han sido cumplidos.

Por otro lado, he de analizar la dificultad que me ha supuesto el desarrollo de este proyecto. Si bien en un primer momento pensaba que este sería un proyecto sencillo pero largo por la gran cantidad de contenido a realizar, al final ha resultado ser todo lo contrario, pues en ciertos puntos la dificultad ha superado con creces a la imaginada en al comienzo del proyecto, sobre todo en el área de desarrollo de servers y comunicación de datos. En un principio se pensaba que la mayoría de tiempo y esfuerzo se destinarían a la utilización del

motor gráfico para el desarrollo de personajes, la estructura del juego, etc. Sin embargo, el desarrollo del modo multijugador, debido a las pocas facilidades que ofrece el motor para su implementación, ha sido con diferencia la parte más complicada del proyecto.

Si miramos el proyecto en el ámbito universitario, y por lo tanto con una finalidad más centrada en el aprendizaje que en el desarrollo de un producto final, puedo decir que ha sido el proyecto con el que más conocimientos he adquirido desde que empecé las clases en la universidad. Éste era uno de los principales objetivos personales, tal y como se expuso en los motivos para la elección del trabajo. Sin embargo, el aprendizaje ha superado ampliamente cualquier expectativa que pudiera tener en un principio. Sin duda, los dos ámbitos en los que más he aprendido han sido en el desarrollo de servidores web y en las tecnologías utilizadas para la comunicación de datos, siendo ambos los dos puntos que, tal y como he comentado anteriormente, más esfuerzo y dificultad han supuesto, así como en los que en un primer momento no pensaba que fuera a necesitar una gran inversión de tiempo y de esfuerzo. Observando el proyecto como un todo, me han quedado claras las dificultades de trabajar en un cierto ámbito sin las facilidades que te ofrece un profesor experto en él y cuyo trabajo consista en transmitir esos conocimientos a sus alumnos. Sin embargo, tras este proyecto también estoy seguro de que soy capaz de solventarlas por mí mismo dedicando el esfuerzo suficiente a ello.

Además de todo esto, una de las partes que más he disfrutado del proyecto ha sido la prueba pública que se realizó con usuarios que, a pesar de no conocerme, decidieron probar el juego. Todos ellos fueron muy simpáticos y amigables, proponiendo además muchos añadidos y cambios que, a pesar de no poder entrar en el alcance del proyecto, son posibles ideas para ampliar el juego y mejorarlo en un futuro.

Por último, me gustaría agradecer a toda las personas que han ayudado a desarrollar Brawlchemy: desde profesores hasta amigos y compañeros de clase, pero especialmente a Mikel Villamañe, el tutor y supervisor del proyecto, por toda la ayuda que ha ofrecido con las dudas que me han surgido a la hora de realizar el proyecto y la memoria. De igual manera, gracias a todos los lectores de este documento, pues espero que la lectura haya sido interesante y amena.

Bibliografía

[1] "Ccoo — convenio colectivo tic." https://www.ccoo-servicios.es/archivos/sms/BOE-A-2018_ConvenioTIC.pdf. Acceso: 2019-04-15.

[2] "Over half of activision blizzard's $7.16 billion yearly revenue came from microtransactions." https://www.techspot.com/news/73230-over-half-activision-blizzard-716-billion-yearly-revenue.html. Acceso: 2019-04-16.

[3] "Ubisoft makes more money from microtransactions than digital game sales." https://gamerant.com/ubisoft-microtransaction-sales/. Acceso: 2019-04-16.

[4] "Top 10 most played moba — gamers decide." https://www.gamersdecide.com/articles/top-10-most-played-mobas. Acceso: 2019-03-13.

[5] "Game info — league of legends." https://euw.leagueoflegends.com/en/game-info/. Acceso: 13-03-2019.

[6] "Og vs el - eu lcs 2015 summer w4d2 - origen vs elements." https://www.youtube.com/watch?v=ZLIu2Pyoj0c. Acceso: 16-03-2019.

[7] "Dota 2 on steam." https://store.steampowered.com/app/570/Dota_2/. Acceso: 16-03-2019.

[8] "Game overview — heroes of the storm." https://heroesofthestorm.com/es-es/game/. Acceso: 14-03-2019.

[9] "Blizzard trabajó en una versión de heroes of the storm para xbox one." https://www.3djuegos.com/noticia/153458/0/heroes-of-the-storm/xone/. Acceso: 16-03-2019.

[10] "Awesomenauts - the free 2d moba." https://www.awesomenauts.com/home/. Acceso: 14-03-2019.

[11] "Awesomenauts - the 2d moba." https://store.steampowered.com/app/204300/Awesomenauts__the_2D_moba/. Acceso: 16-03-2019.

[12] "Official smite wiki." https://smite.gamepedia.com/Smite_Wiki. Acceso: 14-03-2019.

[13] "Review: Smite - a godly moba that gives league of legends a run for its money." http://www.nintendolife.com/reviews/switch-eshop/smite. Acceso: 16-03-2019.

[14] "Reddit." https://www.reddit.com/. Acceso: 2019-06-5.

A. Anexo A: Casos de uso Extendidos

En este anexo se muestran los casos de uso extendidos de Brawlchemy. En ellos se mostrara una descripción del caso de uso, los actores involucrados, las precondiciones y postcondiciones, los requisitos el caso de uso y el flujo de eventos del mismo. Tras ello se podrán ver las interfaces del juego que participan en dicho caso de uso.

A.1. Registro

Descripción: El usuario anónimo puede registrarse en el sistema rellenando un formulario con los datos necesarios.

Actores: Usuario anónimo.

Precondiciones: Ninguna.

Requisitos no funcionales: Ninguno.

Flujo de eventos:

1. El usuario hace clic en el botón «register» en la pantalla inicial.
2. El usuario introduce los datos requeridos y pulsa el botón «register».
   - *a*) Si los datos son correctos, se vuelve a la pantalla inicial.
   - *b*) Si alguno de los datos no son correctos, se muestra un error junto al campo correspondiente. Estos errores se muestran en la figura 64.

Postcondiciones: En caso correcto se ha creado un nuevo usuario en el juego.

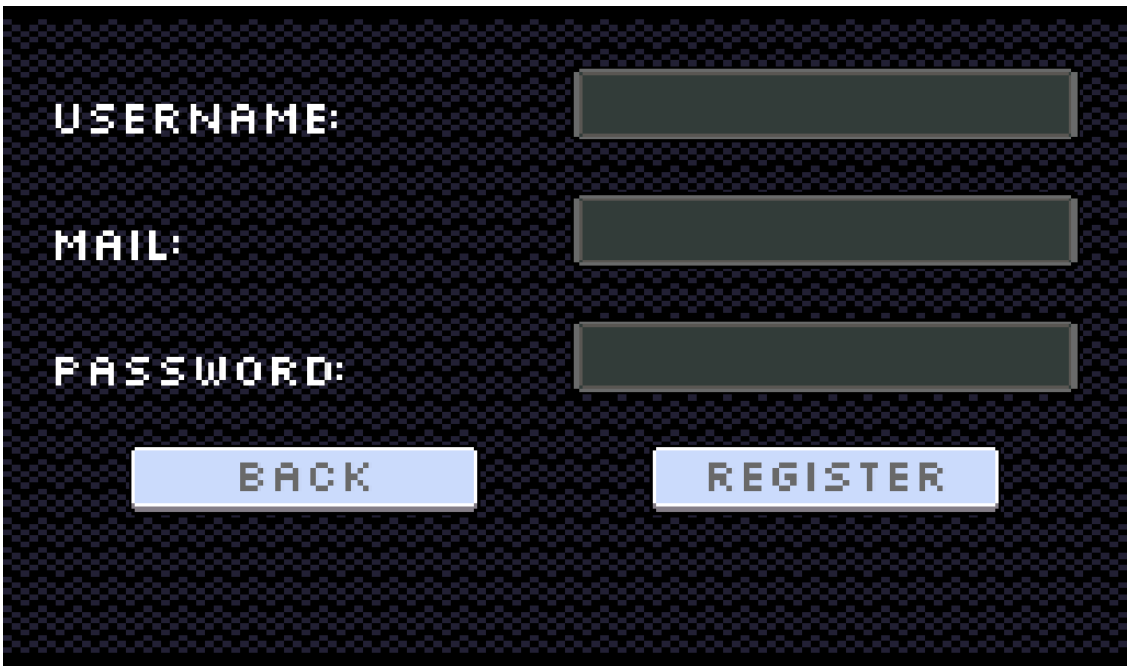


Figura 63: Interfaz del menú de registro

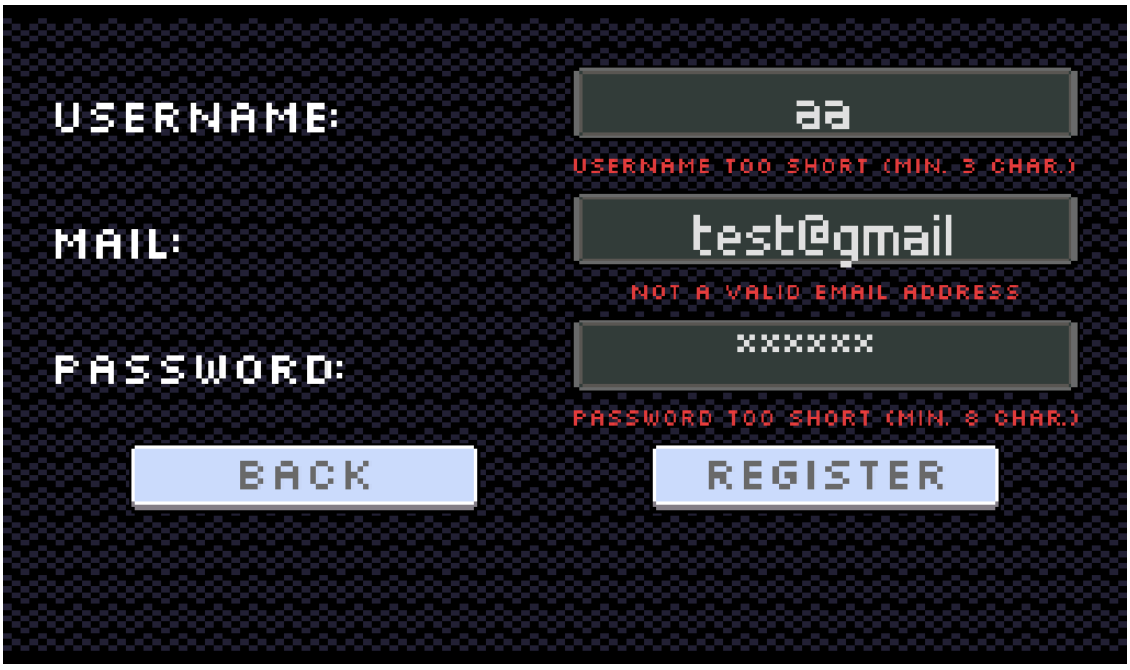


Figura 64: Interfaz del menú de registro cuando las credenciales no son correctas

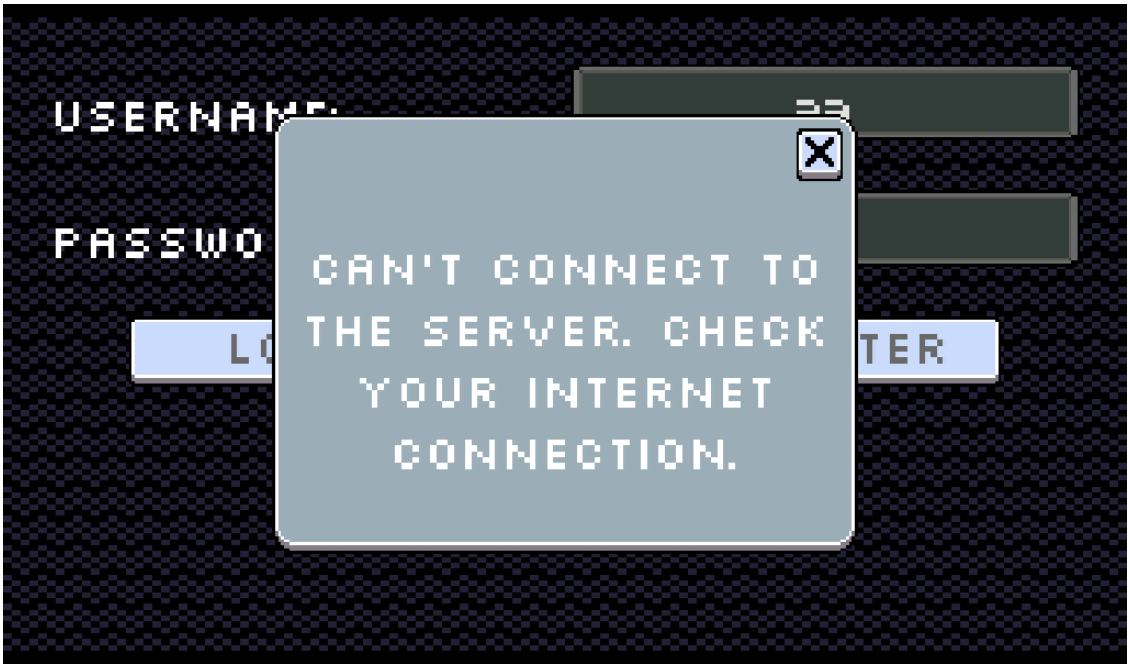


Figura 65: Interfaz del menú de registro cuando se da un error de conexión

A.2. Login

Descripción: El usuario anónimo puede iniciar sesión en el sistema rellenando un formulario con sus datos personales.

Actores: Usuario anónimo.

Precondiciones: Ninguna.

Requisitos no funcionales: Ninguno.

Flujo de eventos:

1. El usuario introduce los datos requeridos y pulsa el botón «login».
   - *a*) Si los datos son correctos, se accede al menú principal del juego como usuario registrado.
   - *b*) Si alguno de los datos no son correctos, se muestra un mensaje de error. Éste se puede ver en la figura 67.

Postcondiciones: En caso correcto el actor anónimo se transforma en actor registrado y se accede al menú principal del juego.

Figura 66: Interfaz del menú de login

Figura 67: Interfaz del menú de login cuando las credenciales no son correctas

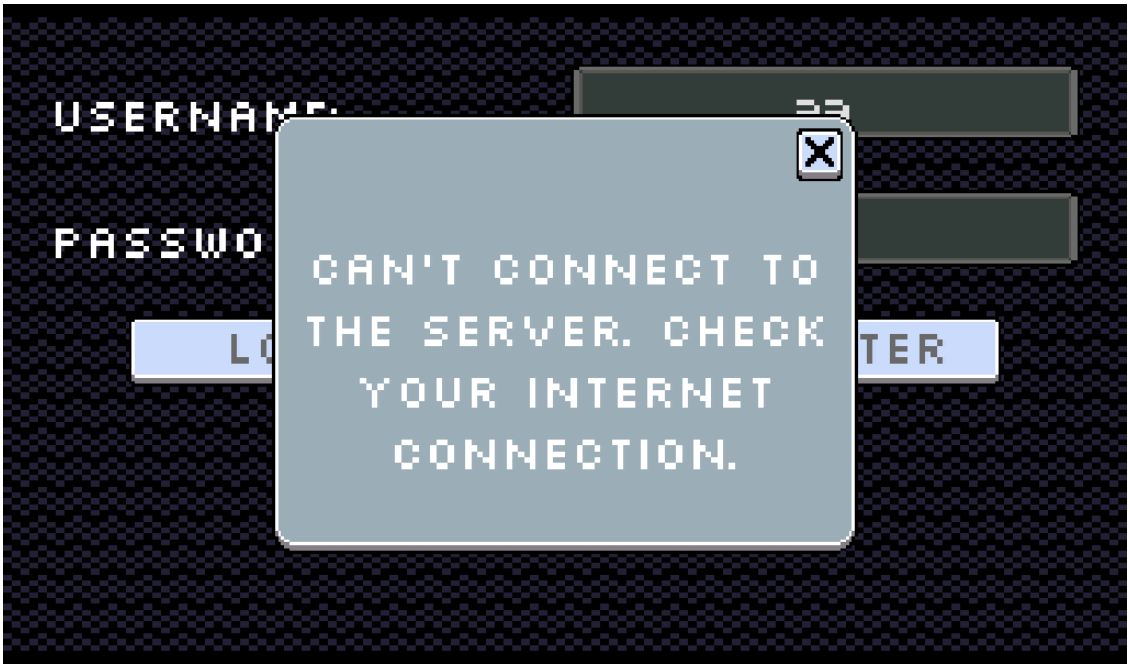
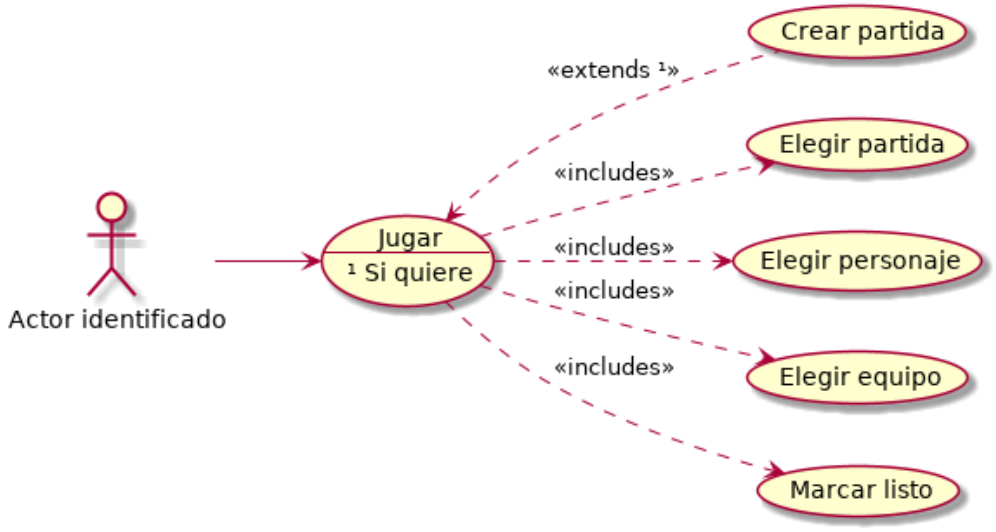


Figura 68: Interfaz del menú de login cuando se da un error de conexión

Figura 69: Interfaz del menú principal

A.3. Jugar



Descripción: El usuario puede unirse a una partida, configurar su personaje y equipo, y comenzar a jugar.

Actores: Usuario identificado.

Precondiciones: Ninguna.

Requisitos no funcionales: Ninguno.

Flujo de eventos:

1. El usuario selecciona el botón «Play» en el menú principal.
2. Se accede a una pantalla en la que se listan las partidas disponibles.
    a) Si lo desea, el usuario puede crear una nueva partida.
3. El usuario elige una partida.
4. Se accede a la pantalla del lobby.
5. El usuario selecciona su personaje (se asigna uno de forma predeterminada).
6. El usuario elige su equipo (se asigna uno de forma predeterminada).
7. El usuario selecciona el botón «ready». Cuando todos los jugadores del lobby hayan seleccionado esta opción, se accede a la partida.

Postcondiciones: Se inicia la partida según los parámetros que los jugadores que participan en ella hayan seleccionado (personajes y equipos).

Figura 70: Interfaz del menú de la lista de partidas disponibles

Figura 71: Interfaz del menú de creación de una nueva partida

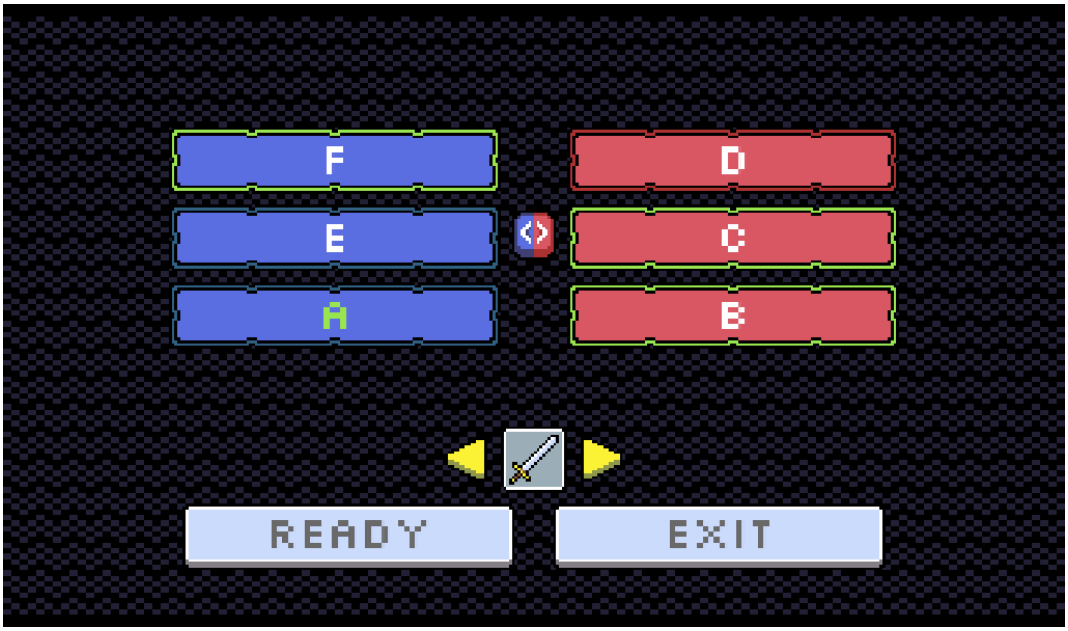
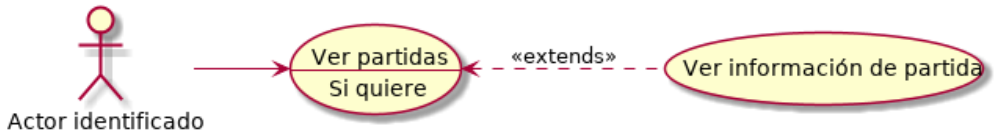


Figura 72: Interfaz del lobby de la partida

A.4. Ver partidas



Descripción: El usuario puede ver información acerca de las partidas en las que ha participado.

Actores: Usuario identificado.

Precondiciones: Ninguna.

Requisitos no funcionales: Ninguno.

Flujo de eventos:

1. El usuario pulsa el botón «Games».
2. Se accede a una pantalla en la que se muestra al usuario una lista con las partidas que ha jugado.
3. Si el usuario selecciona una partida, se muestra información detallada acerca de esta.

Postcondiciones: Se accede a la pantalla que lista las partidas disponibles, y si el usuario lo desea, se muestra la información de aquella partida que elija.

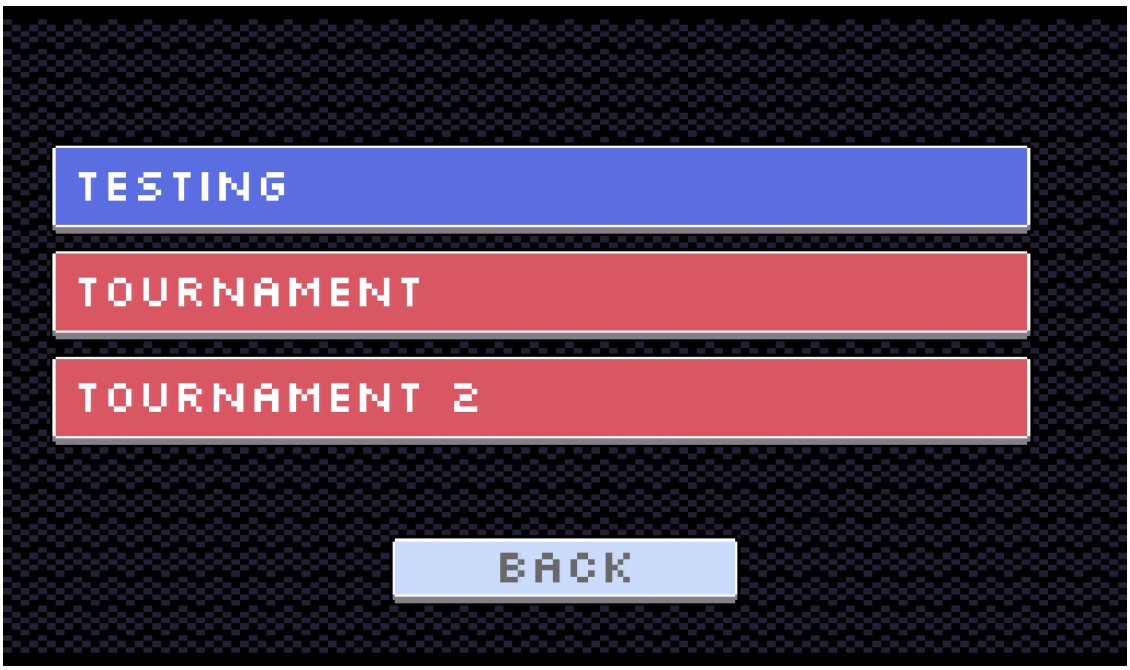


Figura 73: Interfaz del menú del historial de partidas

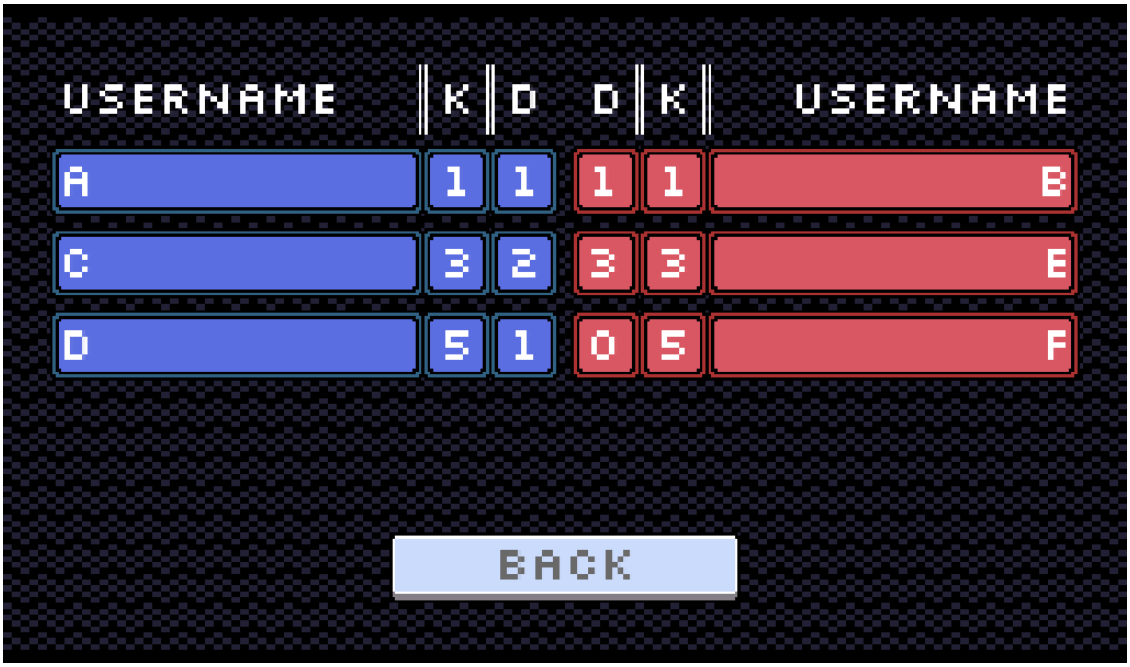


Figura 74: Interfaz del menú de la información extendida de una partida

A.5. Ver controles

Descripción: El usuario puede ver los diferentes controles del juego.

Actores: Usuario identificado.

Precondiciones: Ninguna.

Requisitos no funcionales: Ninguno.

Flujo de eventos:

1. El usuario pulsa el botón «Controls».
2. Se accede a una pantalla en la que se muestran al usuario los controles del juego.

Postcondiciones: Se accede a la pantalla de los controles del juego.

Figura 75: Interfaz del menú de la información extendida de una partida

B. Anexo B: Diagramas de secuencia

En este anexo se muestran los principales diagramas de secuencia de Brawlchemy. Estos se han dividido en dos partes: los que tienen relación con los menús del juego y los que representan acciones que se dan dentro del propio juego entre sus elementos.

B.1. Menús

En esta sección se pueden ver los diagramas de secuencia correspondientes a los menús del juego.

B.1.1. Inicio de sesión

A continuación se muestra el diagrama de secuencia del caso de uso del inicio de sesión.

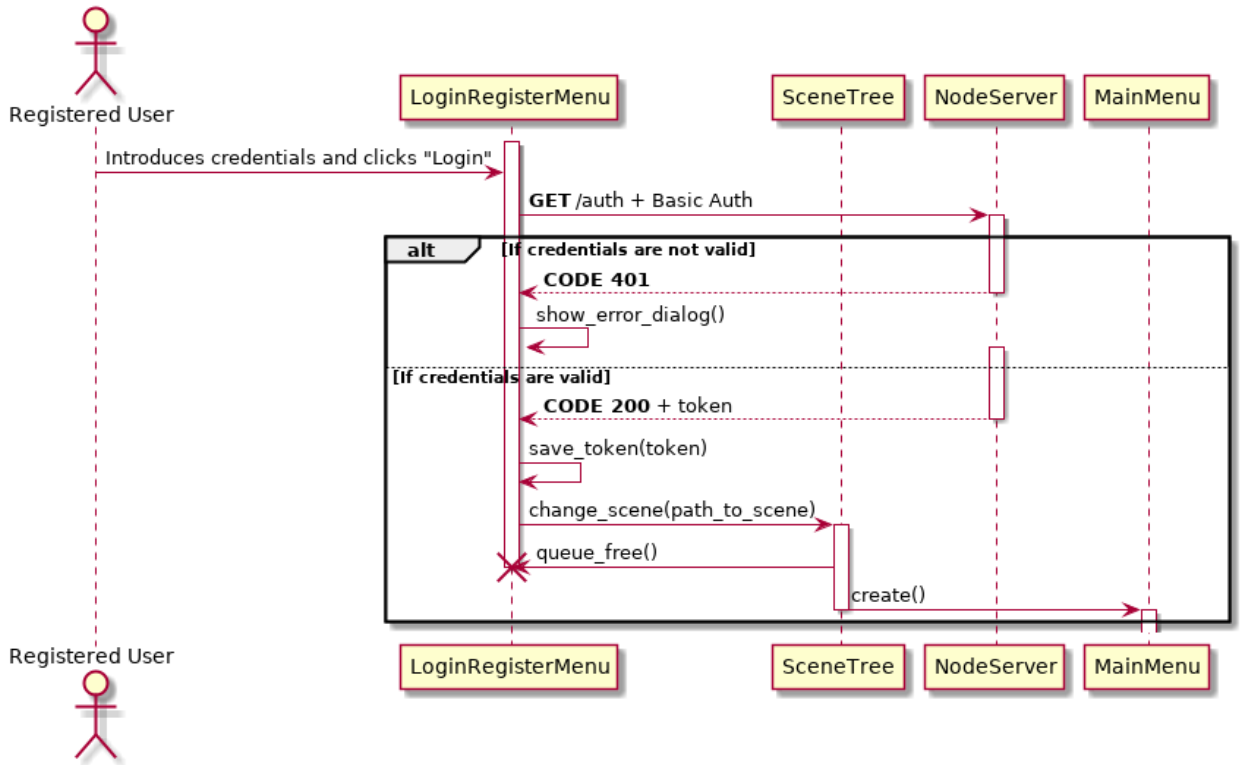


Figura 76: Diagrama de secuencia del inicio de sesión

Como se puede observar en el diagrama, el usuario introduce sus credenciales y pulsa el botón correspondiente al login. Tras ello, esas credenciales son enviadas al servidor, donde

se comprueba su validez. En caso de que no sean válidas se devuelve un código de error y se muestra un mensaje al usuario. En caso de que sean válidas se devuelve un código de éxito y un token. Este token es almacenado, y tras ello se cambia la escena a la correspondiente al menú principal.

B.1.2. Registro

A continuación se muestra el diagrama de secuencia del caso de uso del registro.

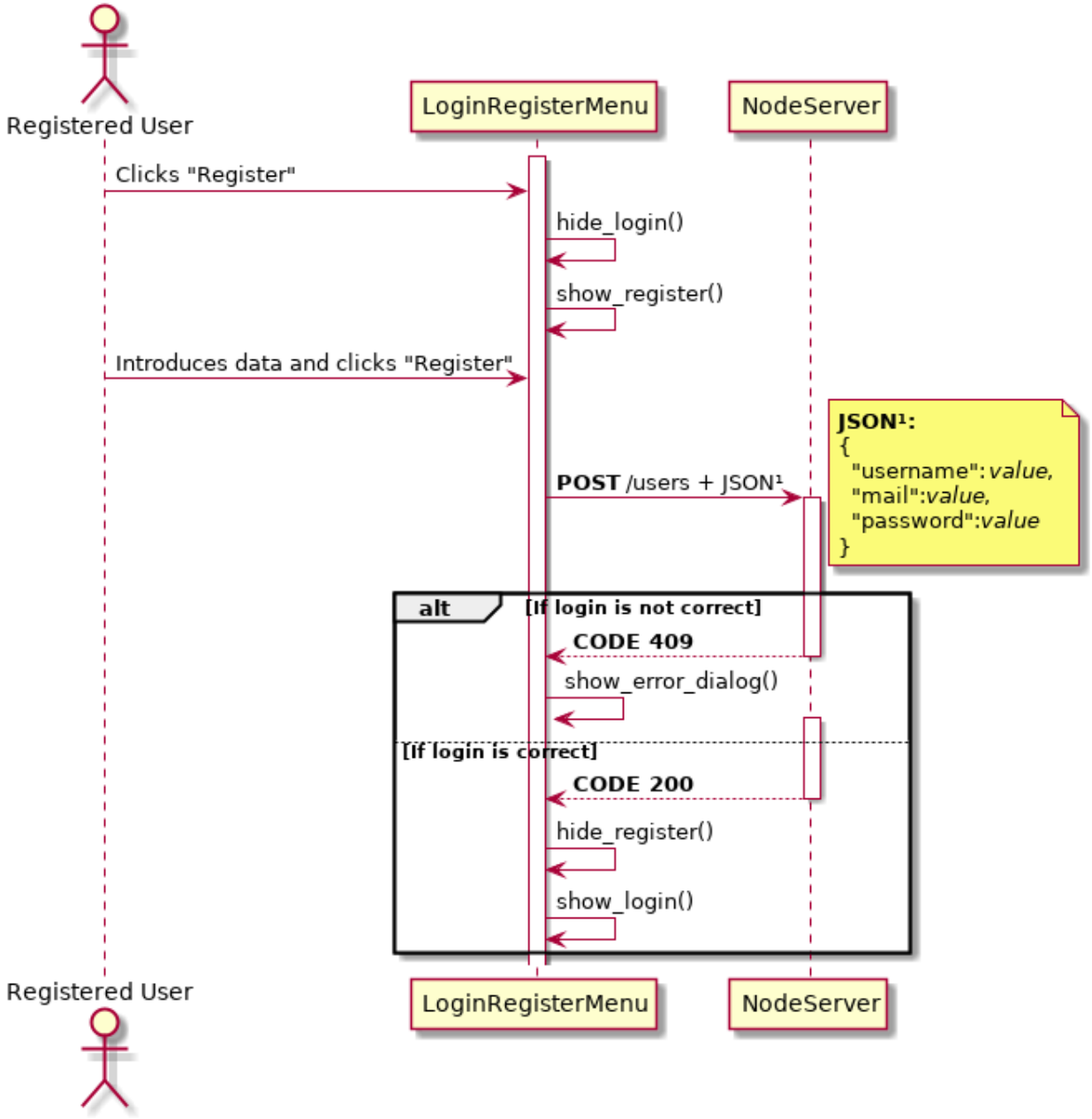


Figura 77: Diagrama de secuencia del registro

En un primer momento el usuario pulsa el botón de registro, tras lo que se oculta el menú de login y se muestra el de registro. A continuación el usuario introduce los datos

necesarios y pulsa el botón para registrarse. En ese momento se realiza una petición HTTP en la que se envían los datos que el usuario ha introducido. En caso de que los datos sean incorrectos se devuelve un código de error y se muestra un mensaje al usuario. En caso contrario, el servidor devuelve un código de éxito. Tras recibir esta confirmación de que el registro se ha realizado correctamente, se oculta el menú de registro y se muestra de nuevo al usuario el menú de login.

B.1.3. Jugar

A continuación se muestra el diagrama de secuencia del caso de uso de acceder a una partida.

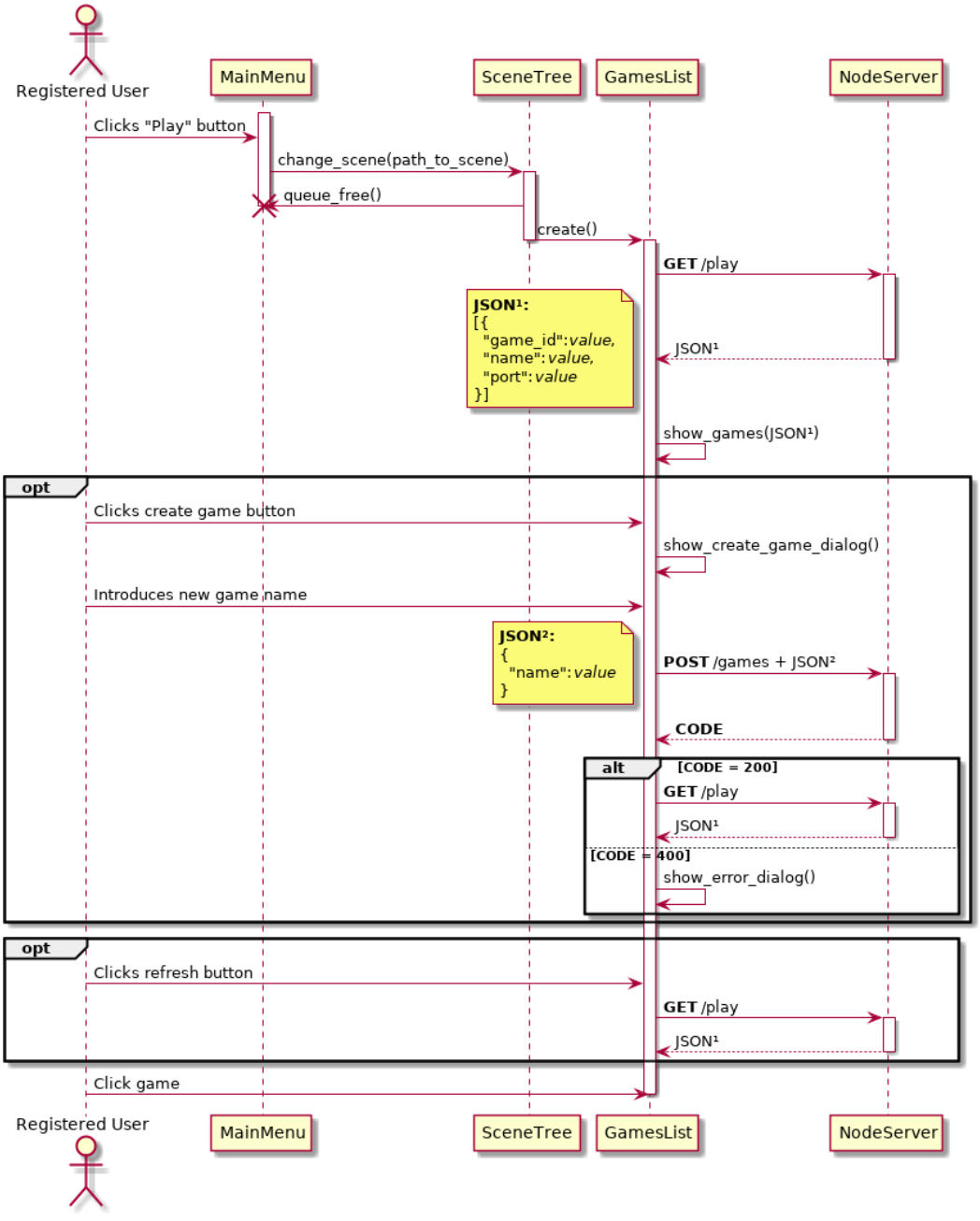


Figura 78: Diagrama de secuencia del acceso a una partida (1)

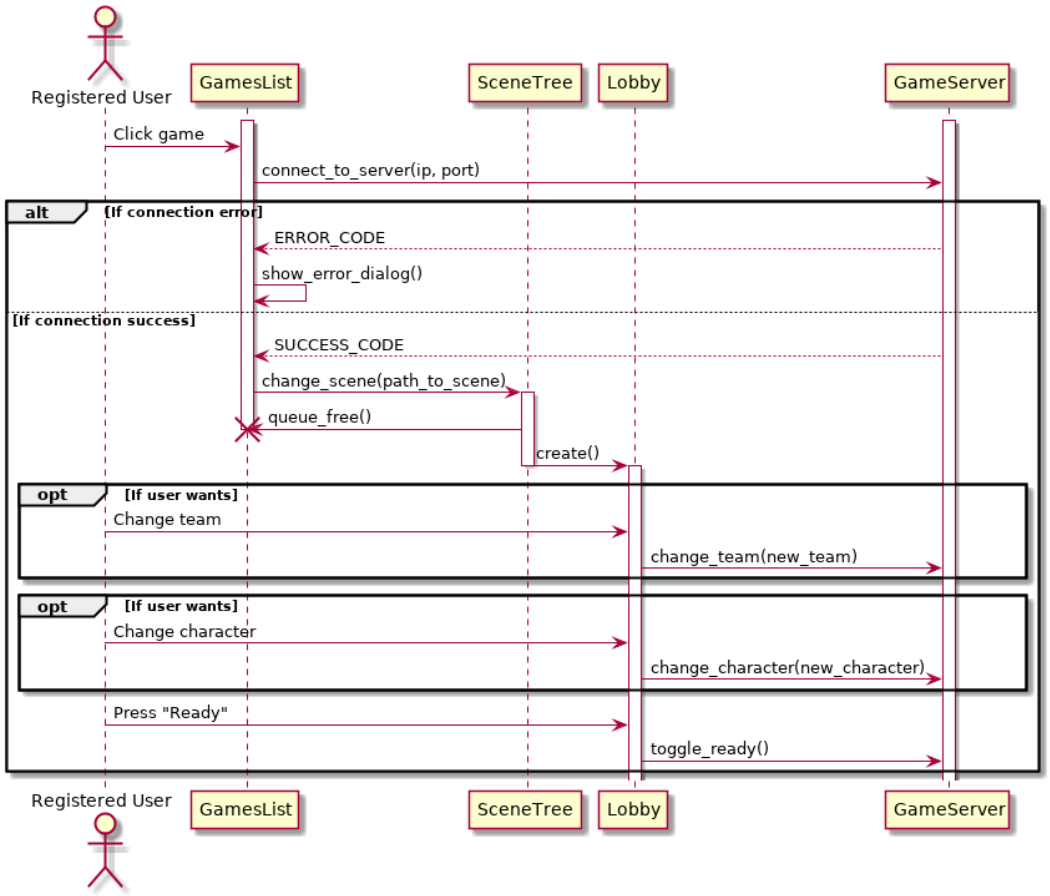


Figura 79: Diagrama de secuencia del acceso a una partida (2)

Desde el menú principal el usuario selecciona la opción para comenzar a jugar. En ese momento se cambia la escena a la de la lista de partidas disponibles. Desde esta escena se realiza una petición HTTP al servidor para obtener un JSON que contenga las partidas disponibles. Al recibirlas, estas son mostradas al usuario en forma de lista. En ese momento el usuario puede realizar dos acciones opcionales: crear una nueva partida o actualizar la lista de partidas.

En caso de que seleccione el botón para crear una nueva partida, se le mostrará un diálogo en el que deberá introducir un nombre para la partida y pulsar el botón que confirma la creación. Tras esto, mediante una petición HTTP se enviará la información al servidor, quien creará la partida y devolverá un código. En caso de éxito, se recarga la lista de partidas repitiendo la misma petición inicial, tras lo cual la partida creada estará disponible. En caso de error, se muestra un mensaje de error.

En caso de seleccionar la opción de recargar la lista de partidas, se vuelve a realizar la petición inicial para mostrar cualquier posible cambio en las partidas.

Por último el usuario selecciona una de las partidas disponibles, tras lo cual se intenta conectar al servidor de la partida seleccionada. En caso de que no se pueda conectar al servidor, se muestra un mensaje de error. En caso contrario, si la conexión se realice con éxito se cambia la escena al lobby de la partida.

A partir de este punto, se ha de tener en cuenta que las acciones que toma cada usuario del lobby deben ser comunicadas al servidor, siendo este quien las comunica al resto de usuarios conectados al servidor y por tanto en el lobby. Esto se ha simplificado en el diagrama, pues sólo aparecen los envíos de información al servidor, ya que la recepción de datos se produce de forma asíncrona.

En este momento el usuario puede realizar dos acciones opcionales: modificar el equipo en el que se le ha colocado automáticamente, o elegir un personaje distinto al asignado. En ambos casos se comunica este cambio al servidor.

Por último, el jugador pulsa el botón de «listo», comunicando al servidor que está listo para empezar la partida.

B.1.4. Historial de partidas

A continuación se muestra el diagrama de secuencia del caso de uso de la visualización de partidas jugadas.

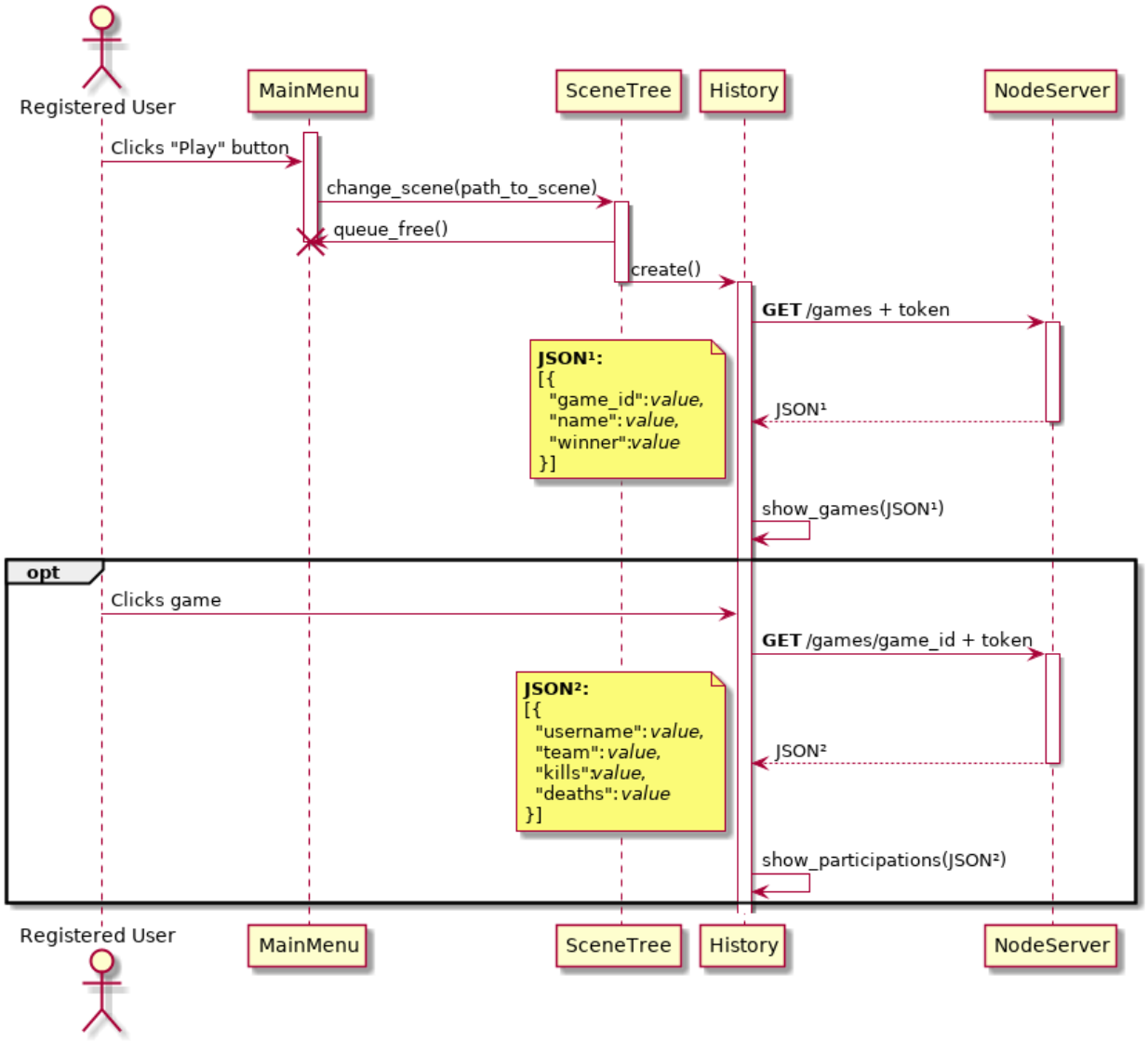


Figura 80: Diagrama de secuencia del historial de partidas

Desde el menú principal el usuario selecciona la opción para ver las partidas jugadas. En ese momento se cambia la escena a la de partidas jugadas. Desde esta escena se realiza una petición HTTP al servidor para obtener un JSON que contenga las partidas jugadas por el usuario. Al recibirlas, mediante un método se muestran en forma de lista. En ese momento el usuario puede seleccionar una partida, para ver más información de ella. Al hacerlo, se

realiza una nueva petición HTTP al servidor, devolviendo este una lista de las participaciones de la partida seleccionada. De forma similar a como se hizo con las partidas, mediante otro método se mostrarán al usuario las participaciones de forma gráfica.

B.1.5. Controles

A continuación se muestra el diagrama de secuencia del caso de uso de la visualización de los controles.

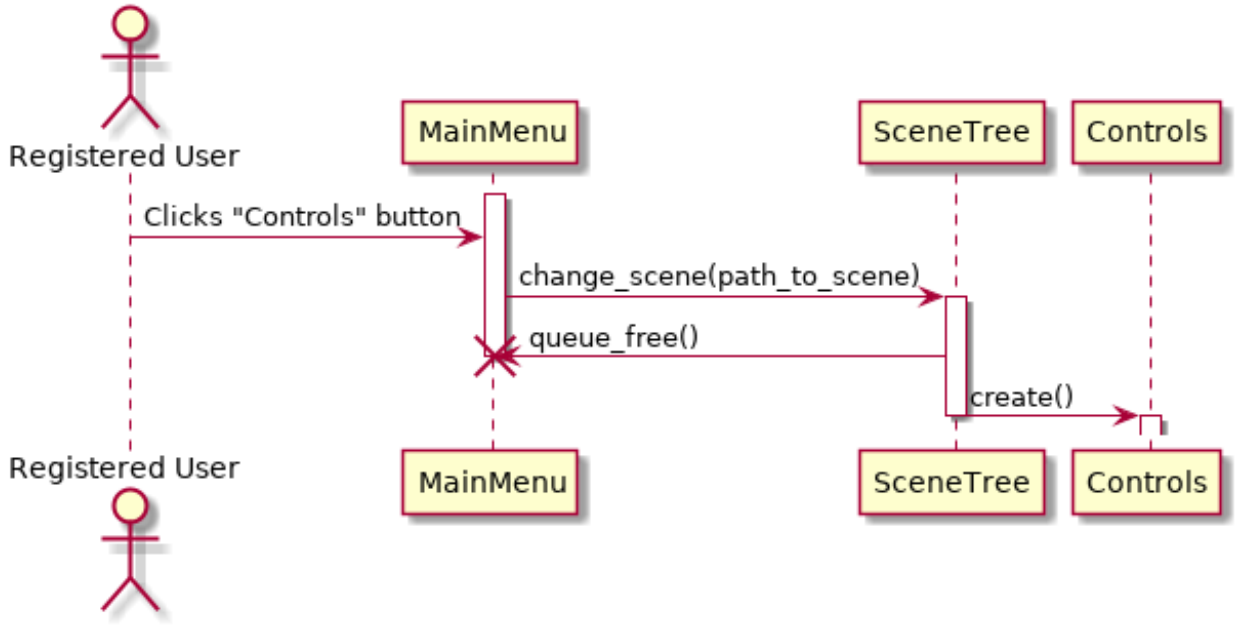


Figura 81: Diagrama de secuencia del menú de controles

Desde el menú principal el usuario selecciona la opción para ver los controles. En ese momento se cambia la escena a la de los controles, en la que se muestra al usuario una imagen que contiene la información necesaria para jugar a Brawlchemy.

B.2. Juego

De la misma forma que con el diagrama de escenas que se realizó en el apartado 5.2, es difícil realizar un diagrama de secuencia al uso con la estructura de un proyecto en Godot debido a la forma en la que se organiza el proyecto. En este apartado se ha realizado una simplificación de las operaciones que se realizan dentro de cada diagrama para que se muestren de una forma entendible pero a la vez representativa.

Además, se han dividido los diagramas de cada escena en dos diagramas distintos, pues la funcionalidad de éstas se realiza tanto de forma síncrona (método «_process») como asíncrona. Como se podrá ver a continuación, los diagramas síncronos se encuentran completamente rodeados por un bucle loop, indicando que toda la funcionalidad interior será procesada en cada frame del juego. Por otro lado, los diagramas asíncronos sirven para representar la interacción entre la escena que representan y señales o eventos provenientes de otras escenas. Cabe destacar que en los diagramas asíncronos los eventos que llegan a la escena principal no se deben tomar en ningún caso como secuenciales a pesar de que se encuentren representados de forma consecutiva, pues cada una de estas operaciones puede ocurrir de forma independiente del resto.

B.2.1. Nexos

A continuación se muestra el diagrama de secuencia asíncrono de los nexos.

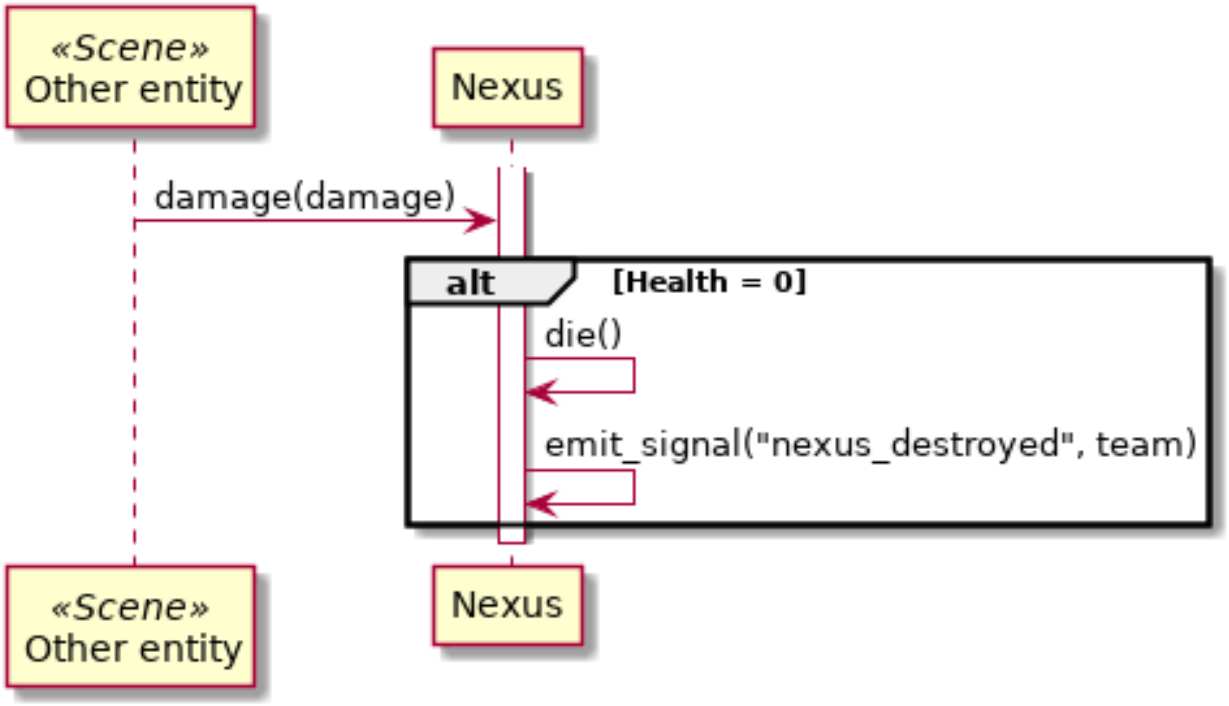


Figura 82: Diagrama de secuencia asíncrono de los nexos

Como se puede ver, la única funcionalidad de los nexos es recibir daño por parte de jugadores enemigos. En caso de que tras recibir este daño la salud del nexo sea 0, este es destruido y emite una señal comunicando esta destrucción, así como el equipo del nexo destruido.

B.2.2. Torres

A continuación se muestran los diagramas de secuencia de las torres.

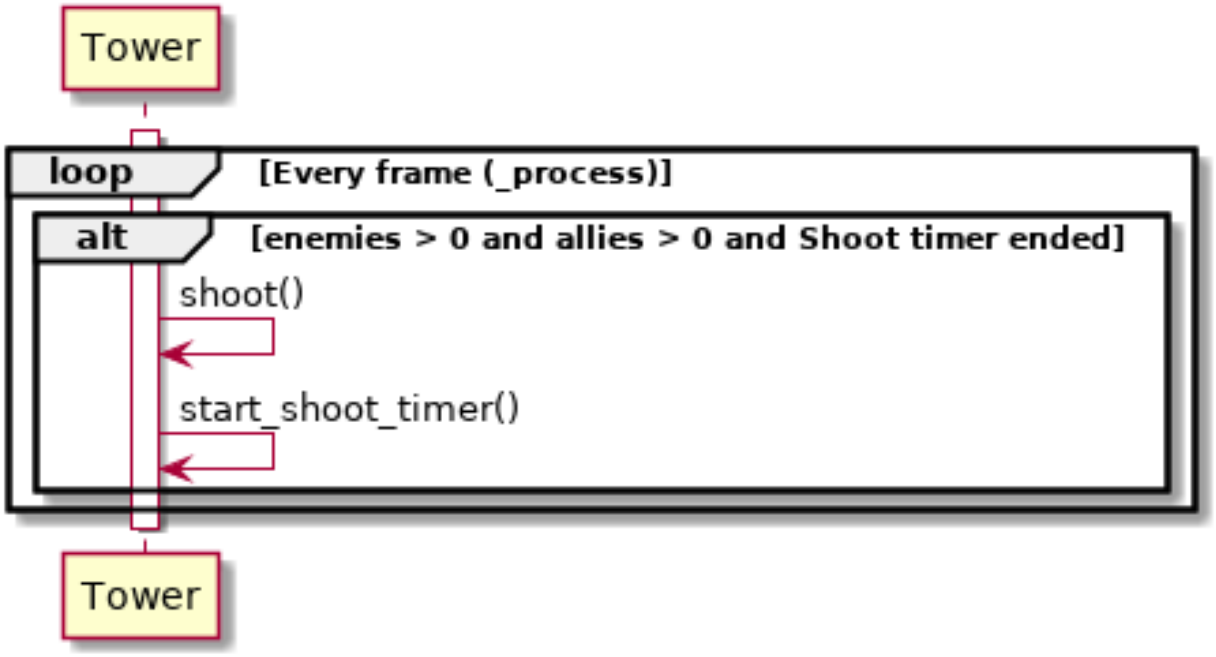


Figura 83: Diagrama de secuencia síncrono de las torres

En la figura 83 se ve el diagrama de secuencia síncrono de una torre. En él, la torre comprueba si existen tanto enemigos como aliados alrededor, así como si ha pasado un tiempo desde el último disparo. En caso de que se cumplan estas condiciones, la torre dispara a uno de los enemigos. Tras ello, inicia el temporizador que marcará el momento en que puede volver a disparar.

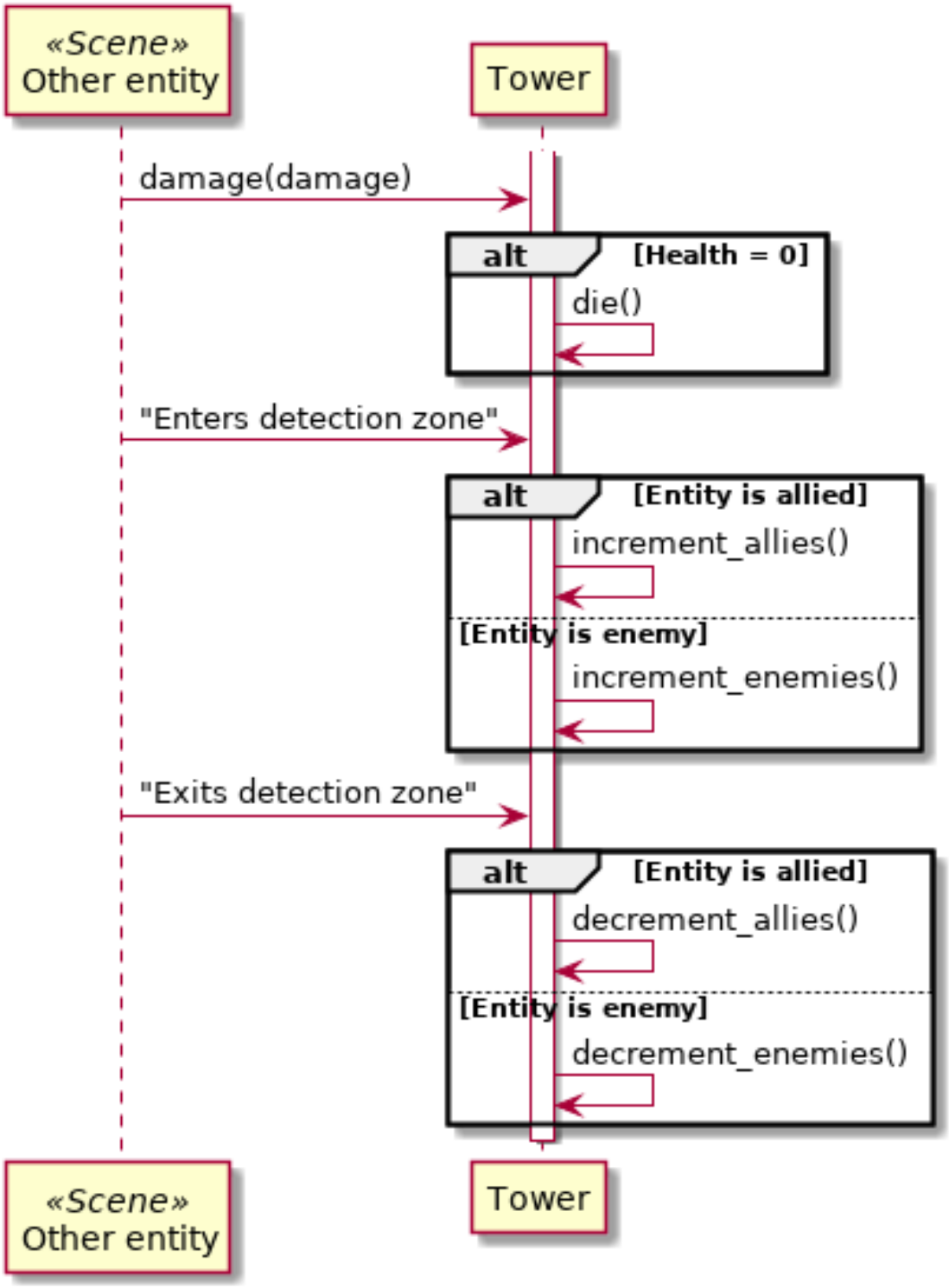


Figura 84: Diagrama de secuencia asíncrono de las torres

En la figura 84 se ve el diagrama de secuencia asíncrono de una torre. Otras escenas pueden interactuar con la torre de tres maneras diferentes.

En primer lugar, y de igual forma que los nexos, una torre puede ser directamente dañada por un jugador enemigo. En caso de que tras recibir el daño indicado su salud sea 0, la torre es destruida.

Además, la torres dispone de un nodo que detecta el momento en que una escena entra o sale de un área definida alrededor de éste. En caso de que entre, se comprueba si es un personaje aliado o enemigo y se incrementa el contador correspondiente. En caso de que una escena salga, de nuevo se comprueba si es aliada o enemiga y se decrementa el contador correspondiente.

B.2.3. Jugadores

A continuación se muestran los diagramas de secuencia de los jugadores.

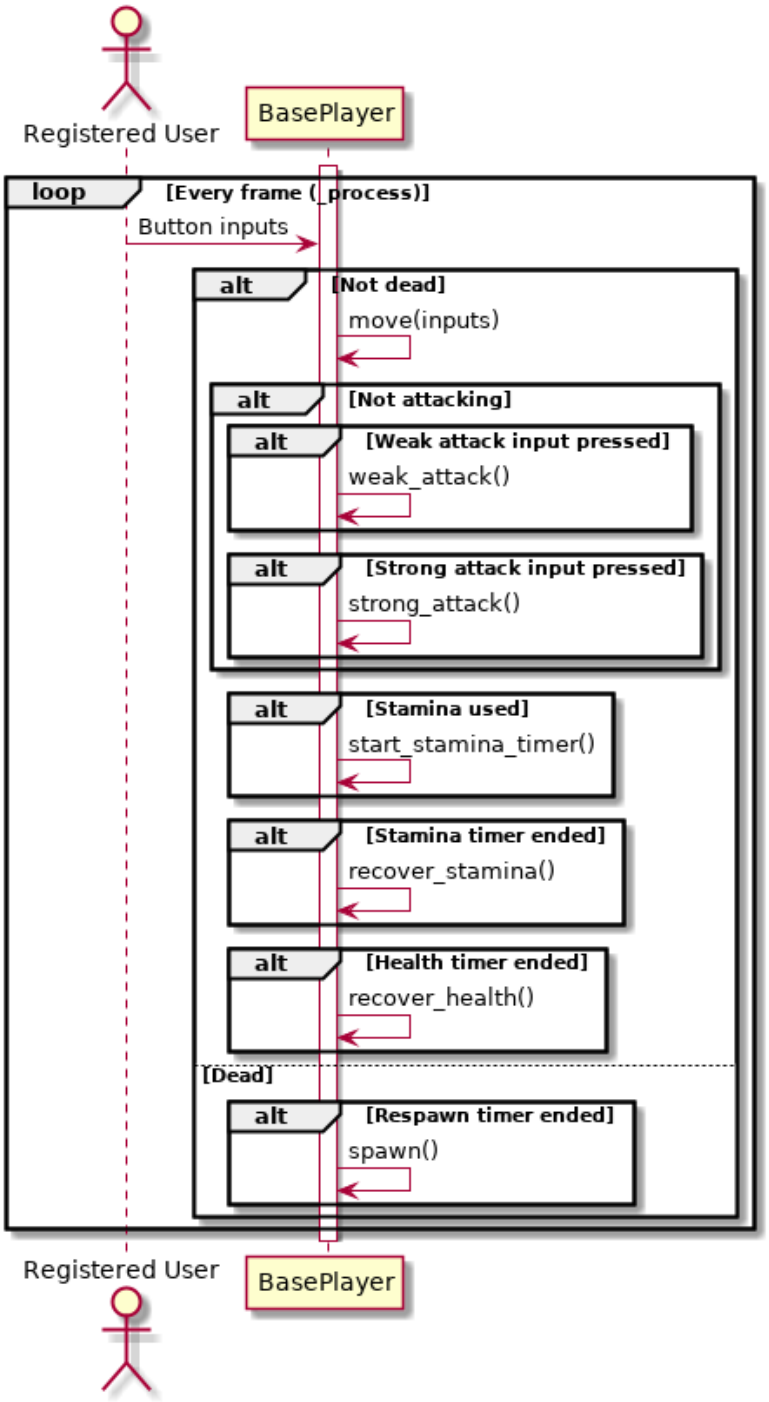


Figura 85: Diagrama de secuencia síncrono de los jugadores

En la figura 85 se ve el diagrama de secuencia síncrono de un jugador. En un primer momento, este siempre recibe los inputs del usuario. Después de ello, la funcionalidad varía en función de si el jugador está muerto o no.

En caso de que no esté muerto, se utilizan los inputs del usuario para mover al personaje. Si éste ha pulsado una tecla de ataque y el personaje no se encuentra atacando en ese momento, se inicia un ataque. Además, en caso de que haya sido necesario utilizar energía para alguna de las acciones, se inicia un temporizador para la recuperación de energía.. Por último, si ha pasado un tiempo desde que el personaje ha utilizado energía esta se recupera progresivamente. De la misma forma, si no ha recibido daño en un tiempo, la salud también se recupera.

En caso de que sí esté muerto, se comprueba si el temporizador de reaparición ha terminado. En caso de que haya terminado, el jugador reaparece en su punto de aparición correspondiente.

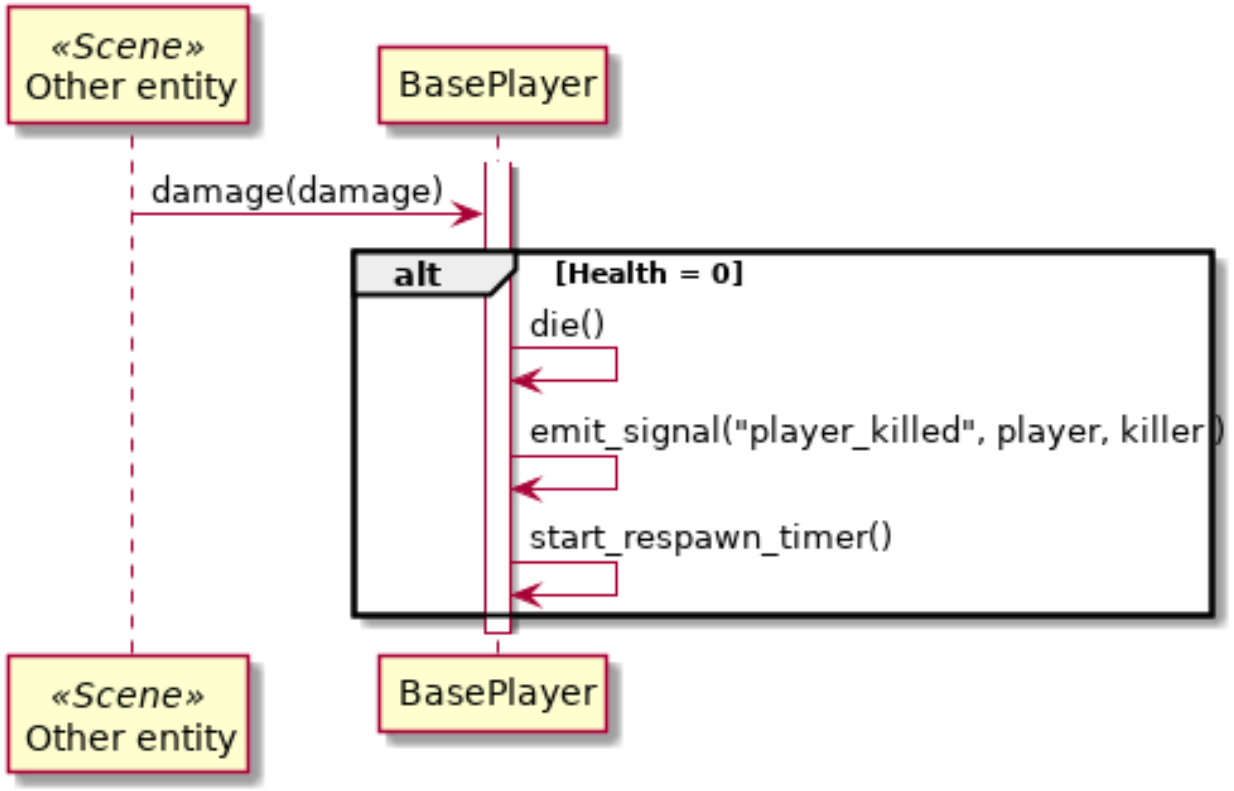


Figura 86: Diagrama de secuencia asíncrono de los jugadores

En la figura 86 se ve el diagrama de secuencia asíncrono de un jugador. La única forma en que otras escenas pueden interactuar con el personaje es haciéndole daño. En caso de que se le haga daño, el personaje pierde la cantidad de vida indicada. Si tras ello su salud se encuentra a 0, el personaje muere, se emite una señal avisando de ello y, por último, se inicia un temporizador que marcará el momento en que puede reaparecer.

B.2.4. Mapa

A continuación se muestra el diagrama de secuencia asíncrono del mapa del juego.

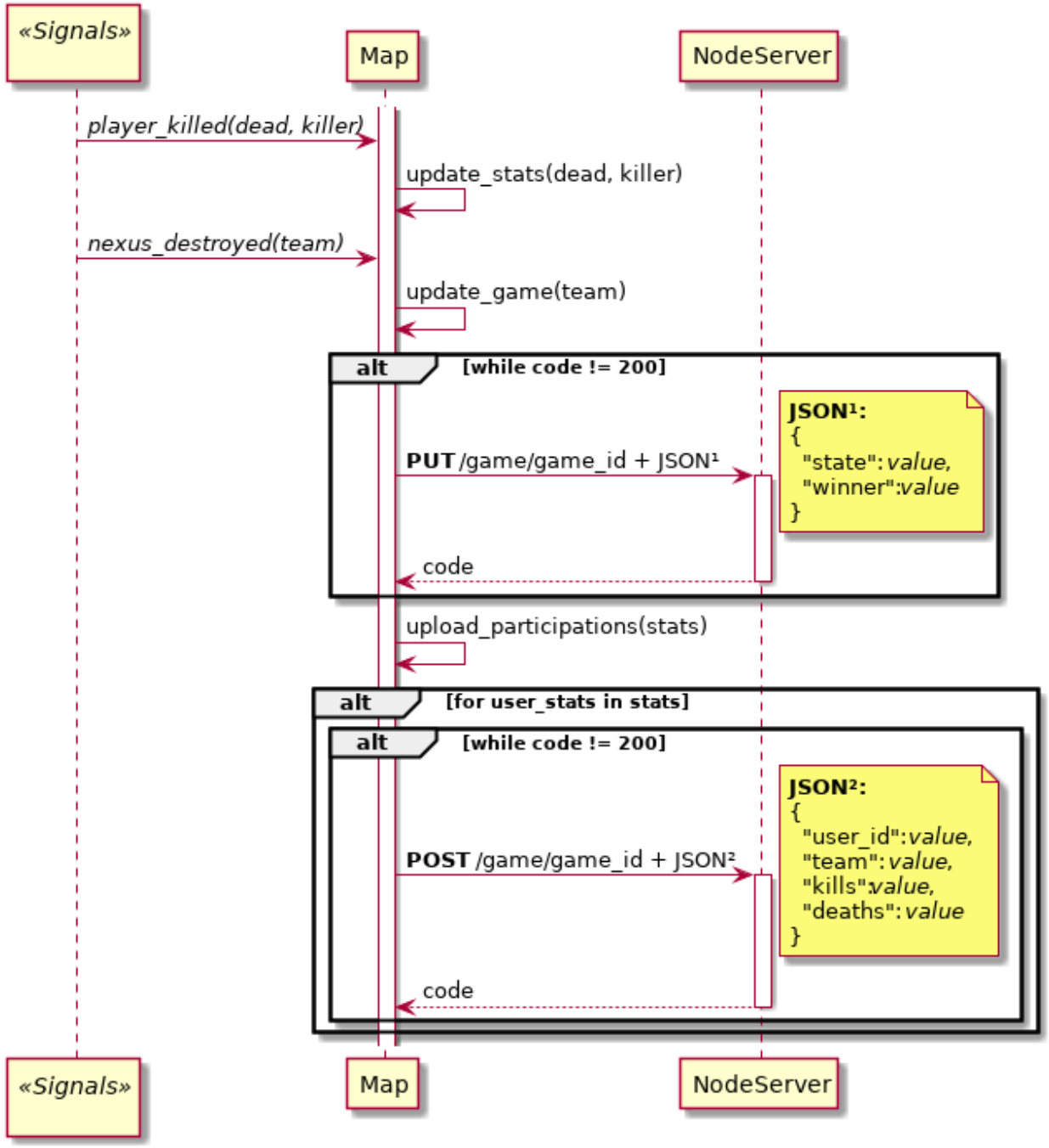


Figura 87: Diagrama de secuencia asíncrono del mapa

El mapa inicia todas sus funcionalidades en base a las señales que otras escenas emiten.

Por un lado, cuando un personaje emite la señal que indica su muerte, el mapa actualiza las estadísticas del jugador que ha matado y el jugador que ha muerto.

Por otro lado, en caso de que uno de los nexos emita una señal comunicando su muerte, el mapa puede proceder a terminar el juego y enviar las estadísticas necesarias al servidor de la base de datos. En una primera petición se actualiza el estado de la partida y se indica el ganador de ésta. En segundo lugar, se recorren las estadísticas de los jugadores y se añade cada una de ellas como una participación en la partida.

C. Anexo C: Introducción a Godot

En este anexo se explican los conceptos necesarios para comprender el funcionamiento de Godot y la construcción de juegos en este motor gráfico.

C.1. Nodos

Los nodos son la base para la construcción de juegos, siendo en cierta manera el equivalente de un objeto en un lenguaje de programación convencional. Existe una gran variedad de nodos, teniendo cada uno una función específica: mostrar una imagen en pantalla, reproducir audio, mostrar modelos en 3 dimensiones, medir tiempos, etc. Sin embargo, todos los nodos tienen unas características principales:

- Posee un nombre propio.
- Tiene una serie de propiedades y variables modificables.
- Se puede extender para añadir funcionalidades.
- Puede procesar sus funciones en cada frame.
- Puede tener otros nodos como hijos.

Utilizando estas características es posible construir diferentes estructuras y dotar a los nodos de las características que se deseen. De las que se muestran en esta lista, la última es especialmente importante, pues mediante el anidamiento de diferentes nodos se forman árboles. Mediante árboles se pueden utilizar diferentes nodos para que, mediante la combinación de sus funcionalidades, se implementen funcionalidades más complejas. Es aquí donde entran en juego las escenas.

C.2. Escenas

Una escena es una combinación de nodos jerárquicamente en forma de árbol. Las escenas con tres características principales:

- Sólo tiene un nodo raíz.
- Se puede guardar como fichero (escena empaquetada o *packed scene*).
- Puede ser instanciada.

Las escenas son la base de cualquier proyecto en Godot, pues en el momento de la ejecución del juego el usuario estará viendo una escena. Si bien es posible disponer de un proyecto creado mediante una única escena, en una situación normal siempre será preferible dividir este proyecto en escenas más pequeñas, para reducir de esta forma su tamaño y complejidad.

De la misma forma que los nodos se pueden organizar en forma de árbol, a una escena se puede añadir una escena empaquetada como hijo. Este es el proceso que en Godot se conoce como instanciación. Es aquí donde entra en juego la división en escenas más pequeñas, pues una misma escena puede ser instanciada múltiples veces dentro de otra, tanto manualmente desde el propio editor como por medio del código creado en las escenas, conocido como script.

Generalmente las escenas que se instancian mediante el editor son aquellas que permanecen de forma continua en el mapa. Por otro lado, las escenas que se instancian mediante código suelen ser aquellas que se creen debido a las acciones del usuario.

Por ejemplo, si se estuviera creando un videojuego similar a Super Mario Bros, la estructura de la escena podría ser de la siguiente forma:

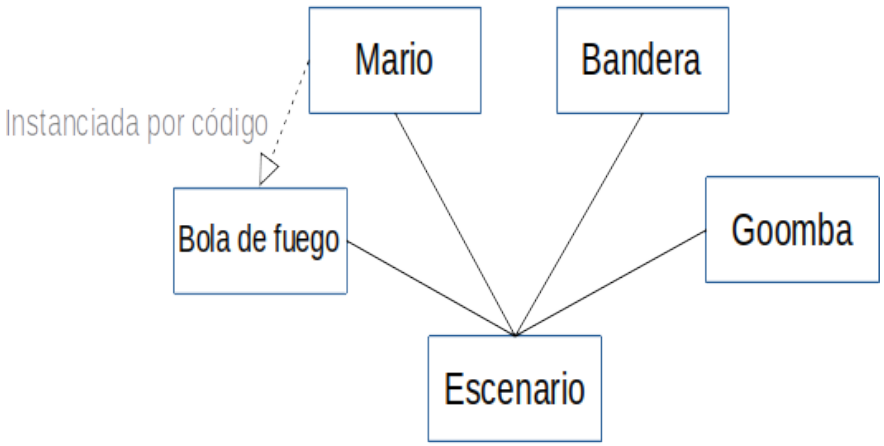


Figura 88: Estructura de las escenas en un juego similar a Super Mario Bros

Como se puede ver, es la escena del escenario la que contiene el resto de escenas, y sería esta la que el Godot estaría gestionando como escena activa, es decir, sería la que se está mostrando al usuario. El resto de escenas han sido instanciadas desde el editor por el desarrollador, a excepción de la bola de fuego. Esta se crea cuando el jugador pulsa un determinado botón, a lo que Mario reacciona lanzando una bola de fuego, lo que se traduce a la instanciación de la escena «Bola de fuego» por medio del código que gestiona la respuesta al evento del botón. En caso de pulsar éste repetidas veces, Mario podría seguir instanciando más bolas de fuego.

C.3. Scripts

Un script es una pieza de código asociada a un nodo y que extiende la funcionalidad de éste o añade nuevos comportamientos, pues sirve para controlar cómo funciona este nodo y cómo interactúa con otros nodos. De manera automática, un script hereda las propiedades y funciones del nodo asociado. Por ejemplo, un nodo sprite dispone de una imagen que se muestra al usuario. Desde un script asociado a un nodo sprite se puede acceder mediante código a la variable que almacena esa imagen.

El lenguaje que se utiliza principalmente para el desarrollo en Godot se llama GDScript, aunque se pueden utilizar otros lenguajes como C#, C++ o VisualScript (permite programar de forma gráfica, útil para principiantes pero es limitado en sus funciones). Sin embargo, GDScript es la opción indicada para casi cualquier situación, pues es sencillo, casi tan rápido como un lenguaje de bajo nivel como C++, tiene muchos tipos de datos relacionados con cálculos algebraicos (vectores y transformaciones entre otros) y, al ser un lenguaje nativo, dispone de una gran interacción con el motor, como autocompletado de escenas, nodos o señales.

Además, mediante scripts se puede acceder a la función principal de un motor gráfico: el procesamiento de métodos de forma continua durante la ejecución. En todo nodo se dispone del método «_process», el cual es ejecutado por el motor en todos los nodos que lo implementen. Es en este método en donde se debe implementar toda aquella funcionalidad que necesite ser procesada en cada ciclo de ejecución. En el ejemplo que se ha usado antes, este método se debería implementar para la detección de los inputs del usuario y para gestionar el movimiento de Mario, por ejemplo.

C.4. Señales

En Godot las señales son el equivalente al patrón «observador» en otros lenguajes de programación. Mediante ellas un nodo puede mandar un mensaje mientras que todos los nodos que se encuentren conectados a esa señal pueden recibir y reaccionar en base a ella.

La principal utilidad de las señales es que ayudan a mantener independencia entre escenas, pues evita situaciones de dependencia directa al no obligar a una de ellas a esperar que la otra escena siempre exista o se encuentre presente de la misma manera para poder comunicarle ciertos datos a través de un método específico, haciendo que esta segunda escena pueda simplemente conectarse a la primera.

Igual que las escenas, las señales pueden ser conectadas a través del editor o a través de código. La primera de estas opciones suele ser útil cuando las escenas a conectar existen desde un primer momento, mientras que la segunda es más útil cuando una de las escenas se ha instanciado durante la ejecución a través de código.