Facultad

de

Ciencias

TOP RUN: JUEGO DE PLATAFORMAS MULTIJUGADOR

(Top Run: Multiplayer platform game)

Trabajo de Fin de Grado

para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Óscar Santisteban González

Director: Carlos Blanco Bueno

Septiembre - 2020

Contenido

Figuras

Tablas

Código

Resumen

Los videojuegos tienen cada vez más importancia dentro del mundo del entretenimiento. Esta industria no ha parado de crecer desde sus inicios gracias a la gran variedad de géneros y títulos desarrollados, y es ya una de las más lucrativas del sector del ocio. Dentro del amplio abanico de opciones que ofrece, los juegos competitivos multijugador se encuentran actualmente entre los más demandados.

En este documento se expone el diseño y desarrollo de un videojuego multijugador soportado en algunas de las plataformas más populares (PC y dispositivos *Android*). Gracias a su carácter competitivo relajado, unas mecánicas sencillas y una interfaz accesible, se persigue llegar a la mayor cantidad de público posible.

Palabras clave: Videojuego, Unity 3D, Plataformas, Multijugador, Multiplataforma, Juego cruzado, Android.

Abstract

Video games are becoming increasingly important in the world of entertainment. This industry has not stopped growing since its creation thanks to the diversity of developed genres and titles and is already one of the most lucrative in the entertainment sector. Within the wide range of offered options, competitive multiplayer games are currently among the most demanded.

This document describes the design and development of a multiplayer video game supported in some of the most popular platforms (PC and *Android* devices). Thanks to its relaxed competitive nature, simple mechanics, and an accessible user interface, it aims to reach the largest audience possible.

Keywords: Videogame, Unity 3D, Platformer, Multiplayer, Multiplatform, Cross play, Android.

1. Introducción

Nacidos en la década de los 70 con las máquinas arcade, los videojuegos han dejado de ser un producto de nicho para convertirse en una de las mayores y más lucrativas industrias del mundo del entretenimiento. De hecho, su influencia económica es tal que, ya en 2019, los ingresos generados por estos superaron ampliamente a los ingresos generados por las industrias de la música y el cine juntas (1).

Es una realidad que los videojuegos han penetrado en la sociedad actual para quedarse. Ejemplos como *Fornite*, *Candy Crush* o *Minecraft* no son más que la punta de lanza de una cultura que crece de forma constante en número de adeptos. Y así como crece el número y variedad de consumidores, crece también la oferta con títulos extremadamente diversos en cuanto a género, temática, mecánicas y dificultad. Hoy en día es posible encontrar multitud de propuestas que van desde los videojuegos meramente narrativos hasta aquellos que basan su atractivo en proponer al jugador retos terriblemente desafiantes. Pero si algo ha incursionado en esta industria de forma impetuosa en los últimos años han sido los juegos multijugador. Jugar en compañía y enfrentarse no solo a uno mismo sino a otras personas es algo que ha motivado al ser humano desde el principio de su existencia, y los videojuegos han sabido aprovecharlo en cuando las tecnologías de redes y comunicación lo han permitido. Tanto es así, que atendiendo a las categorías más vistas durante el mes de julio de 2020 en la plataforma de *streaming Twitch.tv*, los 10 juegos más vistos cuentan con características multijugador (2), principalmente de carácter competitivo.

Dentro de las múltiples causas que han potenciado la rápida expansión de los videojuegos en las últimas décadas cabe destacar el haber sabido aprovechar la masificación del uso de los dispositivos móviles. La democratización de esta tecnología y un modelo de negocio *free-to-play* han hecho que resulte extremadamente sencillo para casi cualquier persona acceder a una gran variedad de títulos, desde sencillos y clásicos juegos de puzles hasta mastodónticos juegos competitivos capaces de mover torneos con premios de hasta 1 millón de dólares (3).

A la vista de estos datos, si lo que se quiere es concebir un videojuego dirigido a la mayor cantidad posible de público, parece una buena idea decantarse por la creación de uno que esté orientado a la competición online y que sea soportado tanto en PC como en plataformas móviles. Puede tomarse como un buen ejemplo el título *Fall Guys*, que pese a haber sido lanzado el 4 de agosto de 2020, en apenas una semana llegó a lo más alto en *Twitch.tv*, superando a titanes del sector como *League of Legends* (4) gracias a su propuesta multijugador competitiva, desenfadada, directa y accesible.

Este proyecto, por tanto, presentará el diseño y desarrollo de un videojuego de plataformas competitivo, disponible tanto para ordenadores como para dispositivos móviles *Android*, con mecánicas sencillas pero sólidas que le hagan accesible para un amplio público y que permitan expandir su vida útil.

2. Herramientas, tecnologías y *assets*

En este apartado se detallan las herramientas y tecnologías utilizadas para la realización del proyecto.

2.1 Herramientas

Durante el desarrollo del videojuego se ha hecho uso, fundamentalmente, del motor *Unity 3D* junto al entorno de desarrollo *Visual Studio 2017*.

2.1.1 Unity 3D

Unity 3D es un motor de videojuegos multiplataforma desarrollado por *Unity Technologies*. Gracias a su compatibilidad con múltiples programas de diseño profesional como *Blender*, *Maya*, *ZBrush* o *Adobe Photoshop*, no solo es ideal para el desarrollo de videojuegos, sino que su alcance se extiende a otros sectores, como la automoción, animación o arquitectura (5).

Su extensa documentación, así como los recursos que se pueden encontrar en la web, hacen de *Unity* una plataforma ideal para iniciarse en el desarrollo de los videojuegos, tanto en 3D como en 2D. Además, su facilidad para exportar el juego a múltiples plataformas sin apenas modificar código también facilita el desarrollo de este proyecto, que tiene como objetivo plataformas *Android* y PC.

El acceso a una licencia de uso gratuita fomenta comenzar a trabajar con *Unity*, y no es hasta alcanzar unos ingresos superiores a 100.000$ en los últimos 12 meses cuando se vuelve necesario aumentar el plan a uno de pago (6).

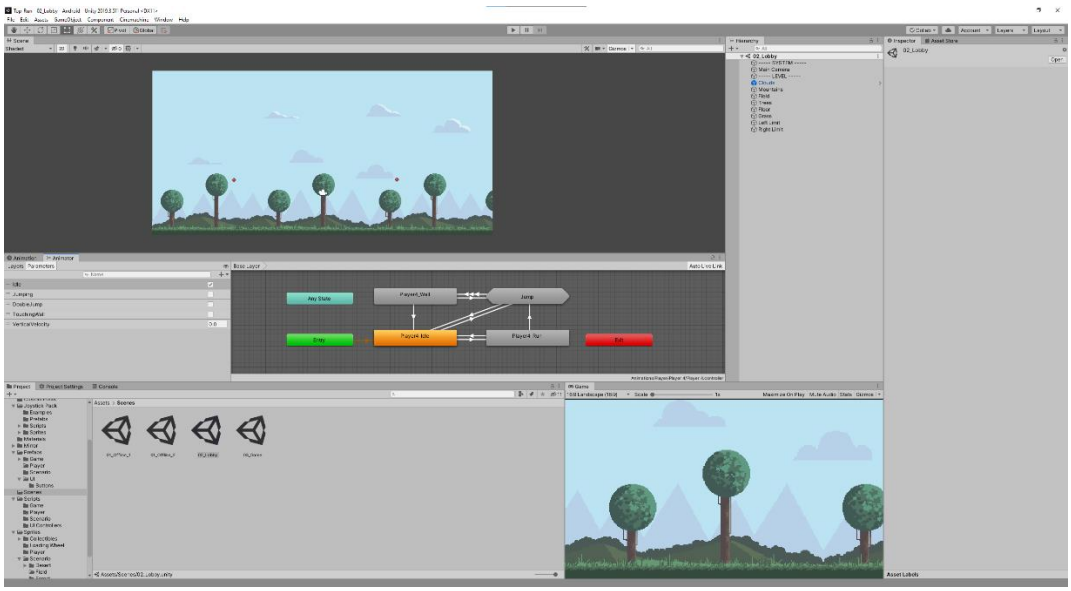


Figura 1. *Interfaz básica de Unity*

Para trabajar con la parte de *scripting*, *Unity* permite emplear diferentes *IDEs*, tales como *Visual Studio 2017* o *MonoDevelop*. Para este proyecto se ha utilizado la primera opción.

2.1.2 Visual Studio 2017

Visual Studio 2017 es un entorno de desarrollo multiplataforma (*Windows*, *MacOS* y *Linux*) compatible con diferentes lenguajes de programación, como *C++*, *C#*, *Java* o *Phyton*. Gracias a su integración nativa con *Unity* y a las múltiples opciones disponibles para agilizar la programación, se postula como una herramienta sólida y práctica.

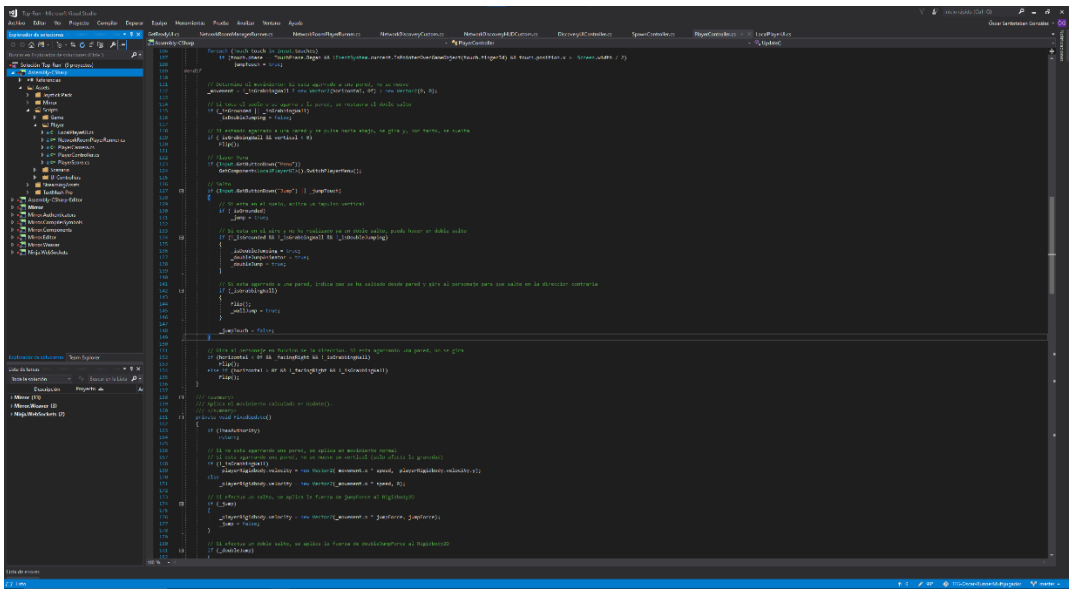


Figura 2. *Interfaz básica de Visual Studio 2017*

2.1.3 Trello

Para la organización personal del proyecto se ha utilizado la herramienta *Trello*, propiedad de la empresa *Atlassian*.

Esta herramienta se basa en la metodología ágil *Kanban* para gestionar las tareas que conforman el proyecto. De este modo, mediante una distribución en tableros y tarjetas, las tareas pasarán por diferentes estados hasta ser completadas. Resulta especialmente útil para, de un vistazo, conocer el estado del proyecto, con las tareas en curso, pendientes y completadas (7).

Esta facilidad esquemática y visual ayuda a los miembros de un equipo a centrarse en el trabajo en curso, además de permitir detectar cuándo el equipo se acerca a un punto de excesiva carga, evitando penalizaciones en la eficiencia.

Aunque en este proyecto no se ha trabajado con un equipo, las ventajas de *Kanban* también han surtido efecto en el trabajo individual.

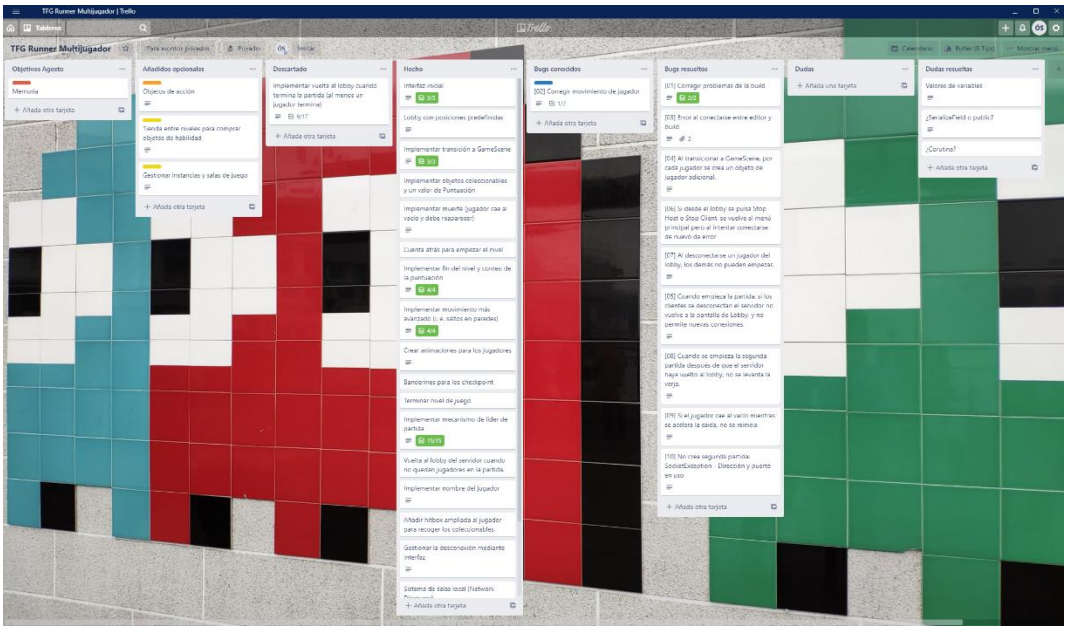


***Figura 3**. Interfaz de Trello*

2.1.4 Krita

Krita es un programa de pintura e ilustración digital *open source* desarrollado por *Krita Foundation*, cuyos miembros pertenecen a la comunidad *K Desktop Environment* (KDE).

Aunque para este proyecto no se han creado desde cero los *assets* gráficos, se ha utilizado *Krita* para el diseño del logotipo y los iconos básicos de Pausa y Continuar.

***Figura 4**. Interfaz personalizada de Krita*

2.2 Tecnologías

2.2.1 Lenguaje de programación C#

Para la parte de *scripting* en *Unity* se emplea el lenguaje orientado a objetos *C#*, desarrollado por *Microsoft.* Esta herramienta también soporta *C++*, pero para un primer

acercamiento a ella es preferible usar *C#*, ya que es más sencillo de aprender y gestiona automáticamente la memoria empleada, facilitando el proceso de desarrollo.

2.2.2 Mirror Networking

Dado que *Unity* carece de *API* propia (contaba con *UNet*, pero está obsoleta y será sustituida por una nueva tecnología aún en desarrollo (8)), es necesario buscar una alternativa externa para desarrollar un juego con capacidades multijugador.

En este caso, la solución escogida ha sido *Mirror Networking*, una *API open source* de alto nivel disponible como *plugin* en la *Asset Store* de *Unity*.

Mirror Networking se construye sobre la obsoleta *UNet*, corrigiendo sus defectos. Utiliza un enfoque bajo el cual el cliente y el servidor ejecutan el mismo código y, a través de etiquetas y directivas específicas, se lleva a cabo la comunicación entre ambos. Este paradigma simplifica el desarrollo, por lo que es una buena elección de cara a un primer acercamiento a la creación de juegos multijugador.

2.2.3 Git

El control de versiones se ha llevado a cabo con *Git*, concretamente a través de la plataforma *GitHub*. Como a este proyecto solo ha contribuido una persona, gran parte de las utilidades que proporciona *Git* han sido desaprovechadas. Sin embargo, sigue siendo útil para registrar y gestionar las distintas versiones por las que ha pasado el juego durante sus distintas etapas de desarrollo.

2.2.4 Android SDK

Para la compilación en *Android*, *Unity* necesita el *Software Development Kit* (*SDK*) de dicho sistema operativo. Esta pieza *software* es la encargada de proporcionar a *Unity las* herramientas necesarias para poder ejecutar el juego en *Android*, y se descarga automáticamente cuando se añade el módulo *Android Build Support* a la versión de *Unity* con la que se esté trabajando.

2.3 *Assets*

Para la parte gráfica se han empleado los *assets* del pack "*Pixel Art Infinite Runner"*, de Eder Muniz, disponible en la página web *Itch.io*. De este pack se han utilizado los *sprites* para los personajes, con sus correspondientes animaciones, los fondos y plataformas de los escenarios y la barrera de inicio del juego.

Para las plataformas de salto impulsado, se ha usado un *sprite* del pack "*Pixel Fantasy Cave"*, de Szadi Art, también disponible en *Itch.io*.

Para las banderas de punto de control se ha usado un *sprite* del pack "*Pixel Game Set"*, de Pixelic Atom, disponible en la página web *Iconfinder.com*; mientras que la bandera de final de juego emplea el *sprite "Mario Flag Pixel Art"*, subido por Liz Valdes a la página web *Kindpng.com*.

Para los coleccionables se han usado *sprites* del pack "*RPG items retro"*, de Emberheart Games, disponible en la página web *Itch.io*.

Para el texto se ha utilizado la tipografía "*Minecrafter"*, de Madpixel Designs, disponible en la página web *Dafont.com*.

El logotipo y los iconos de Pausa y Continuar han sido creados desde cero gracias al *software Krita*, como ya se ha mencionado.

Finalmente, para el *joystick* digital presente en las versiones de *Android* se ha usado el *asset "Joystick Pack"*, disponible de forma gratuita en la *Asset Store* de *Unity*.

3. Metodología

Dada la naturaleza del proyecto, se ha considerado idónea la metodología iterativa incremental. Consiste en iterar sobre ciclos en cascada de análisis de requerimientos, diseño del software, implementación y pruebas, haciendo así que cada característica añadida al proyecto sea implementada y probada antes de avanzar a la siguiente (9).

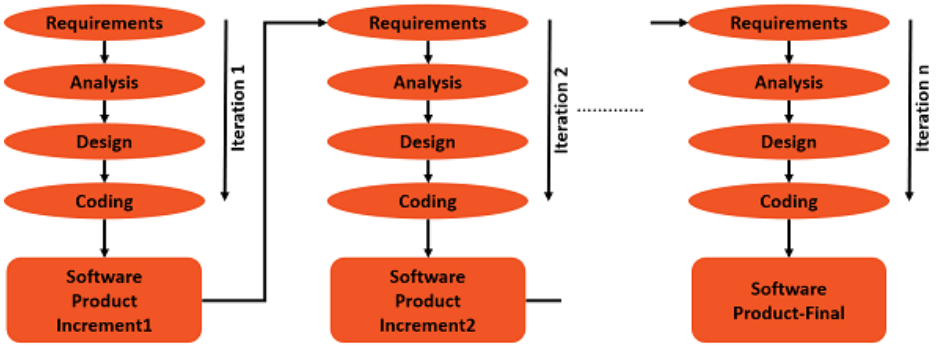


Figura 5. *Diagrama del modelo iterativo incremental*

Como se está desarrollando un videojuego, conviene utilizar esta metodología para poder ir *testeando* los diferentes elementos incorporados. Una mecánica mal diseñada o un elemento aburrido o fuera de lugar puede resultar fatal para el proyecto, obligando a rediseñarlo por completo. Gracias a esta metodología se puede pueden incorporar mecánicas sin miedo a que al final del ciclo de desarrollo resulten inadecuadas.

3.1 Planificación

El proyecto se ha dividido en 3 fases: formación, desarrollo e integración.

3.1.1 Fase de formación

Esta fase se dedicó a familiarizarse con la tecnología e interfaz de *Unity*, así como con las prácticas habituales en el desarrollo de un videojuego. Para ello, se siguió principalmente el tutorial "Introducción a *Unity* para videojuegos 2D", creado por Juan Diego Vázquez Moreno en la plataforma *Domestika.org* (10). En dicho curso se explican los conceptos básicos de un videojuego en 2D de estética *pixel art*, cubriendo tanto el apartado de programación como el de animación a través de lecciones que concluyen en la elaboración de un pequeño proyecto con un personaje principal y múltiples enemigos.

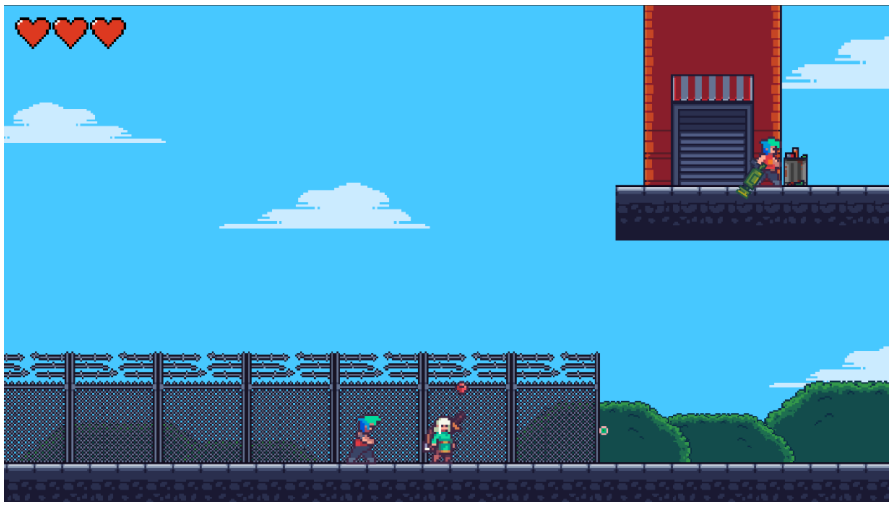


Figura 6. *Proyecto desarrollado durante la realización del curso de formación*

3.1.2 Fase de desarrollo

La fase de desarrollo ha agrupado todas las iteraciones realizadas durante el proyecto. Cada una de estas, como ya se ha mencionado, ha constado de cuatro etapas:

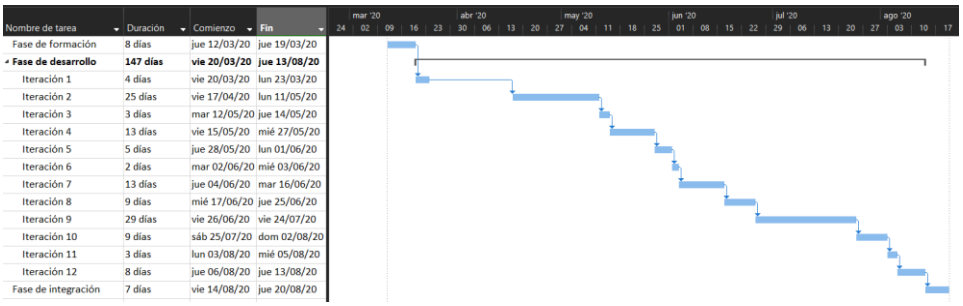
1. **Análisis de requerimientos:** Planificación del conjunto de objetivos a alcanzar durante la iteración, así como de los medios y herramientas necesarios.
2. **Diseño del *software*:** Toma de decisiones enfocadas a cumplir los objetivos definidos en la etapa de análisis. Abarca tanto las decisiones de diseño software como las de diseño visual.
3. **Implementación:** Desarrollo de la lógica y/o elementos visuales necesarios para llevar a cabo los objetivos definidos durante la etapa de análisis, así como su posterior incorporación al proyecto.
4. **Pruebas:** Comprobación del correcto funcionamiento de los elementos añadidos en la etapa anterior, tanto a nivel individual como a nivel de proyecto.

3.1.3 Fase de integración

En esta etapa final se realizaron pruebas bajo distintas condiciones: instancia del juego en diferentes PCs, instancia del juego en diferentes dispositivos *Android* e instancia del juego en diferentes PCs y dispositivos *Android*.

3.1.4 Diagrama de Gantt

La duración de las fases se muestra en el siguiente diagrama de *Gantt* (figura 7).



Nombre de tarea	Duración	Comienzo	Fin
Fase de formación	8 días	jue 12/03/20	jue 19/03/20
Fase de desarrollo	**147 días**	**vie 20/03/20**	**jue 13/08/20**
Iteración 1	4 días	vie 20/03/20	lun 23/03/20
Iteración 2	25 días	vie 17/04/20	lun 11/05/20
Iteración 3	3 días	mar 12/05/20	jue 14/05/20
Iteración 4	13 días	vie 15/05/20	mié 27/05/20
Iteración 5	5 días	jue 28/05/20	lun 01/06/20
Iteración 6	2 días	mar 02/06/20	mié 03/06/20
Iteración 7	13 días	jue 04/06/20	mar 16/06/20
Iteración 8	9 días	mié 17/06/20	jue 25/06/20
Iteración 9	29 días	vie 26/06/20	vie 24/07/20
Iteración 10	9 días	sáb 25/07/20	dom 02/08/20
Iteración 11	3 días	lun 03/08/20	mié 05/08/20
Iteración 12	8 días	jue 06/08/20	jue 13/08/20
Fase de integración	7 días	vie 14/08/20	jue 20/08/20

Figura 7. *Diagrama de Gantt*

4. Análisis de requisitos

A continuación, se detallan los requisitos del proyecto, desglosados en funcionales y no funcionales.

4.1 Requisitos funcionales

Como ya se mencionó en la introducción de este documento, la situación actual del mercado de los videojuegos demanda principalmente proyectos multijugador de carácter competitivo. Por esto, que el software desarrollado cumpla con estas características se convierte en el principal objetivo. Sin embargo, crear un juego competitivo serio y correctamente balanceado requiere de una cantidad de trabajo en diseño y *testeo* que no puede llevarse a cabo en un proyecto del tipo que nos ocupa. Se ha optado por tanto por crear un videojuego más desenfadado y directo, que carecerá de rangos o ligas y en el que los jugadores podrán enfrentarse con el propósito único de obtener la victoria en la partida.

Un ejemplo de este tipo de experiencias está en el recientemente exitoso *Fall Guys*, también mencionado en la introducción. Este juego plantea una jugabilidad muy sencilla y abordable que sustenta a un apartado competitivo relajado en el que perder no tiene penalizaciones más allá de ganar menos experiencia al finalizar la partida. Otros juegos, como la saga *Mario Kart*, repiten este esquema de jugabilidad accesible y tono amigable, pero elaboran unas mecánicas algo más profundas, incentivando el esforzarse por dominar el juego y premiando con victorias a quien lo consigue.

Tomando lo descrito como punto de partida, toca decidir cómo va a ser el apartado de jugabilidad. Dado que otro de los objetivos fundamentales es construir un juego accesible para todos los públicos, los controles irán en consecuencia. Las dos mecánicas más básicas de los videojuegos son, sin duda, correr y saltar. Este esquema viene repitiéndose desde 1985, cuando el clásico de *Nintendo Super Mario Bros*. para la *Nintendo Entertainment System* redefinió y popularizó el género de las plataformas. Sin embargo, tomar como únicas mecánicas aquellas que presentó un título hace 35 años no resulta una buena idea desde un punto de vista de diseño. Así, para dotar al juego de profundidad extra, se añadirán otras mecánicas más modernas, como el doble salto, el agarre a las paredes o el salto impulsado desde las mismas.

Por otro lado, partiendo de las mencionadas mecánicas básicas, parece natural que el objetivo de las partidas sea alcanzar la meta a través de distintos niveles de carrera sobre plataformas. Para dotar de cierta complejidad al juego los niveles constarán de varias rutas para alcanzar la meta. Además, repartidos por ellas habrá distintos coleccionables que incrementarán la puntuación del jugador. Cuanta más habilidad requiera una ruta, más puntuación tendrá que poder obtener aquel que la recorra. Además, la puntuación obtenida incrementará proporcionalmente la velocidad del jugador, de forma que, aunque una ruta sea más larga y compleja, los coleccionables en ella situados permitirán al jugador ir más rápido que los demás y llegar antes a la meta. Sin embargo, siempre que un jugador cometa un error y caiga al vacío reaparecerá al inicio del nivel habiendo

perdido la mitad de su puntuación. Se crea así un sencillo sistema de riesgo-recompensa que favorecerá a aquellos que más dominen el juego.

Por último, queda definir cómo se gestionará el juego *online*. Para este proyecto se ha optado por prescindir de la capa de *matchmaking*, pues *Mirror Networking* carece de ella. Así, para poder llevar a cabo las partidas será necesario que un jugador (jugador *host*) la cree, teniendo el resto de los jugadores que unirse a ella.

Atendiendo a todas estas consideraciones se pueden especificar más formalmente los requisitos funcionales del proyecto:

ID	DESCRIPCIÓN
RF01	El juego permitirá crear partidas *online*, así como unirse a las partidas de otros jugadores.
RF02	Los jugadores que han entrado en la partida aparecerán en una sala de espera.
RF03	Habrá un máximo de cuatro jugadores por partida.
RF04	Si la partida cuenta ya con cuatro jugadores, no se permitirá la entrada de aquellos que intenten conectarse.
RF05	Los jugadores podrán conectarse a la partida únicamente si esta se encuentra en la sala de espera.
RF06	Todos los jugadores podrán elegir un nombre para identificarse en la partida, y no podrán hacer nada más hasta que no hayan fijado uno.
RF07	La partida empezará solo cuando todos los jugadores que la conforman estén listos.
RF08	Solo el jugador que ha creado la partida (jugador *host*) podrá avanzarla hasta el nivel de juego.
RF09	El nivel de juego comenzará con una cuenta atrás para permitir a los jugadores prepararse.
RF10	Si el jugador *host* abandona la partida, esta se deshará y se expulsará al resto de jugadores.
RF11	Si un jugador que no es el jugador *host* abandona la partida, esta debe continuar.
RF12	Los jugadores podrán correr, saltar, efectuar un doble salto, agarrarse a las paredes y saltar desde ellas.
RF13	El desplazamiento en los niveles será lateral e individual, esto es, cada jugador contará con una cámara propia centrada únicamente en él.

RF14	Los jugadores contarán con una puntuación propia que podrán aumentar recogiendo objetos coleccionables.
RF15	Los objetos coleccionables podrán tener diferentes valores de puntuación.
RF16	Los jugadores podrán visualizar su puntuación en todo momento gracias a un cuadro de texto en pantalla, que se actualizará cada vez que se produzca un cambio en dicho valor.
RF17	La velocidad del jugador aumentará con su puntuación.
RF18	Los niveles podrán contar con zonas de caída al vacío.
RF19	Los niveles contarán con puntos de control en los que el jugador reaparecerá cuando caiga al vacío.
RF20	Cuando caiga al vacío, el jugador verá su puntuación reducida a la mitad.
RF21	Cuando un jugador llegué al final del nivel, se bloqueará su movimiento y se mostrará una pantalla con su puntuación final y su posición en la carrera. También se mostrará con un botón para salir de la partida.
RF22	El jugador *host* deberá esperar a que todos los jugadores hayan terminado o abandonado la partida para terminarla.
RF23	Los jugadores podrán desplegar un menú en todo momento que les permitirá desconectarse de la partida.

Tabla 1. *Requisitos funcionales*

4.2 Requisitos no funcionales

Los principales requisitos no funcionales surgen cuando se plantea el público objetivo del producto. Como se quiere llegar a la mayor cantidad de gente posible, el título deberá contar con una interfaz clara e intuitiva. También será imprescindible apuntar al mercado móvil y al de PC, ya que agrupan a la mayoría de los usuarios. Además, para no imponer restricciones de plataforma, deberá ser posible jugar con otras personas independientemente del sistema operativo del dispositivo en el que estén jugando.

Al crear una versión para dispositivos móviles, es importante tener en cuenta el rendimiento. Hoy en día es común encontrar dispositivos con más de 3GB de RAM, por lo que un uso de 2GB de RAM resulta razonable.

Finalmente, para una experiencia fluida será imprescindible que el juego muestre más de 30 fotogramas por segundo.

Con todo esto pueden definirse los requisitos no funcionales del software de manera más precisa:

ID	TIPO	DESCRIPCIÓN	RELEVANCIA
RNF01	Portabilidad	El juego será compatible con dispositivos que corran sistemas operativos *Windows*, *Mac*, *Linux* y *Android*, y los clientes podrán jugar entre sí independientemente de la plataforma.	Muy alta
RNF02	Usabilidad	La interfaz deberá ser simple e intuitiva.	Alta
RNF03	Usabilidad	El juego deberá presentar una estética y unos elementos aptos para todos los públicos.	Alta
RNF04	Rendimiento	El juego consumirá, como máximo, 2GB de memoria RAM.	Alta
RNF05	Rendimiento	El juego mostrará, como mínimo, 30 fotogramas por segundo.	Muy alta

Tabla 2. *Requisitos no funcionales*

5. Diseño e implementación

Para organizar el proyecto se han creado dos capas bien diferenciadas: presentación y negocio. Gracias a esta estructura se pueden separar los elementos de la interfaz con los que va a interactuar el usuario (capa de presentación) de la lógica del programa (capa de negocio).

5.1 Capa de presentación

La intención del juego es ser accesible para todos los públicos, tanto los familiarizados con los videojuegos como los que no lo están. Por tanto, las opciones en la interfaz tendrán que ser pocas y directas.

El videojuego permitirá tanto crear partidas como unirse a las creadas por otros jugadores, con lo cual estás opciones deberán estar muy visibles a través de una interfaz agradable que facilite el juego. Para guiar la navegación, los botones que permitirán avanzar en el sistema (“Jugar”, “Crear Partida”, etc.) tendrán un color amarillo pastel, mientras que aquellos destinados a deshacer alguna acción (“Salir”, “Atrás”) lo tendrán gris oscuro. Este esquema de colores se repetirá a lo largo de todas las interfaces del juego, ayudando al usuario a darle significado a los botones en función de su color.

Así pues, al arrancar el juego, el usuario se encontrará con una sencilla pantalla de inicio con el logotipo del juego y solo dos botones: “Jugar” y “Salir” (figura 8).

Figura 8. *Pantalla de inicio*

El botón de “Jugar” avanza a la creación y selección de partidas, y es claramente el más llamativo, buscando esa intención de atraer la atención del usuario. El de “Salir”, abajo a la izquierda, es más discreto y se encarga de cerrar el cliente del juego y devolver al usuario al escritorio.

Pulsando “Jugar”, se avanza a la interfaz de gestión de partidas (figura 8).

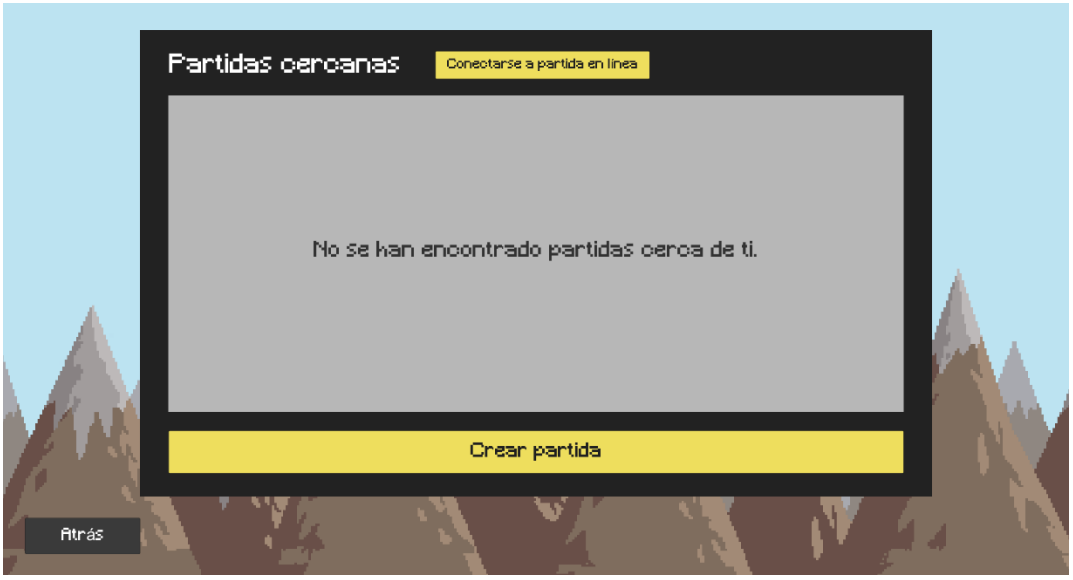


***Figura 9**. Pantalla de creación/selección de partida*

Lo primero que se encuentra el usuario es una ventana con el rótulo "Partidas cercanas" (figura 9). Como el juego tiene capacidades de juego local para disfrutar en compañía, se ha dado mucha importancia a las partidas disponibles bajo la misma red. Así, cuando se crea una partida, esta se anuncia en la red local y, por tanto, aparece en la pantalla del resto de usuarios, mostrando el nombre del jugador *host* y el número de jugadores dentro de esa partida (figura 10). De esta manera, los jugadores que quieran unirse a ella solo tendrán que seleccionarla.

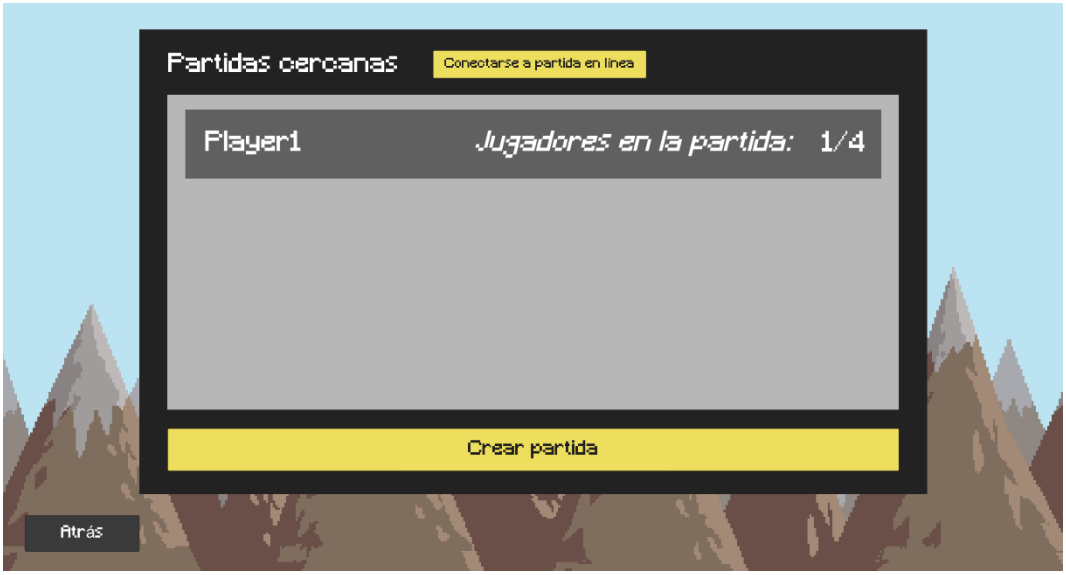


***Figura 10.** Partida local encontrada*

Si varios jugadores hubieran creado una partida, estas se mostrarán en modo lista, y el usuario podrá desplazarse sobre ellas (figura 11). Esta lista se actualiza de forma automática.

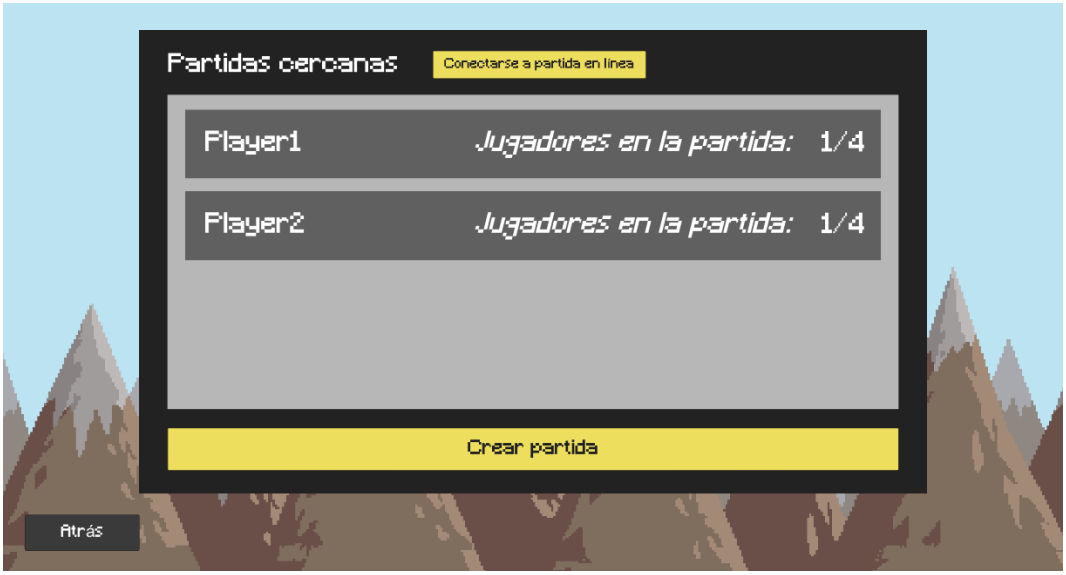


Figura 11. *Múltiples partidas locales encontradas*

Otro elemento gráfico sobre el que el usuario puede prestar atención es el botón "Conectarse a partida en línea", en la parte superior junto al rótulo "Partidas cercanas", agrupando así todas las opciones de conexión a otras partidas. Al pulsar, se despliega una caja de texto en la que se permite introducir la dirección IP del dispositivo que aloja la partida a la que el usuario quiere unirse (figura 12, izquierda). La opción de "Unirse a partida" permanece desactivada hasta que el jugador introduzca algún valor en el campo "IP del servidor" (figura 11, derecha), tras lo cual este se incorporará a la partida. Si el usuario quiere volver atrás, solo tendrá que pulsar en cualquier punto de la pantalla fuera de la ventana.

Figura 12. *Conectarse a una partida en línea*

Si, por el contrario, el usuario no quisiera conectarse a otra partida, sino que quisiera crear la suya propia, encontrará en la pantalla de gestión de partidas el botón "Crear partida", el más grande de todos. Como su nombre indica, este se encarga de crear un nuevo servidor en el que el usuario actúa como jugador *host*.

Cuando un usuario se conecta a una partida, creándola o uniéndose a una ya existente, lo primero que se le muestra es una ventana para que introduzca el nombre que va a usar para representar a su avatar (figura 13). No se le permite llevar a cabo ninguna otra acción hasta que no haya introducido algún texto en este campo y pulsado el botón "Confirmar".

Figura 13. *Ventana de introducción de nombre*

Una vez completados estos pasos, el jugador puede mover su personaje por la sala de espera, en la que se encontrará con el resto de los jugadores de la partida. El nombre que haya elegido se muestra encima de su avatar, por lo que el resto de los miembros de la sala podrán verlo (figura 14, arriba). Cada jugador puede ver su propio botón "Preparado", arriba a la izquierda (figura 14, abajo a la izquierda). Al seleccionarlo, su estado pasará a listo (más adelante se especificará cómo se gestiona este proceso en la lógica del programa) y el botón mostrará el mensaje "Esperando", cambiando su color a verde (figura 14, abajo a la derecha). Si este botón se vuelve a pulsar, el estado del jugador pasará a no listo y el botón se mostrará como al inicio.

Figura 14. *Sala de espera y estados del botón "Preparado"*

La lógica de este botón condiciona el botón "Empezar partida", desactivado por defecto y solo visible para el usuario *host* (figura 15, izquierda). Este botón solo se activa cuando todos los jugadores, incluido el jugador *host*, están en estado listo (figura 15, derecha). Al pulsarlo, todos los jugadores se moverán al nivel de juego.

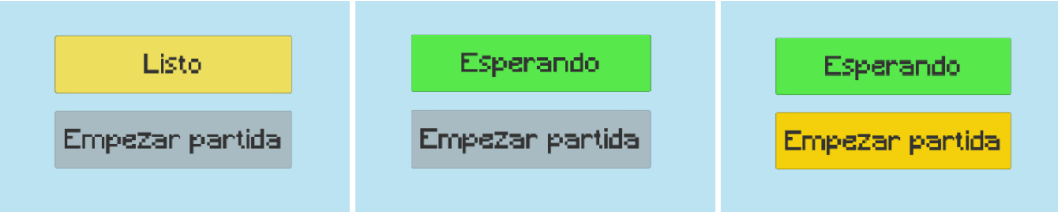


Figura 15. *Estados del botón "Empezar partida"*

Al entrar en el nivel de juego, se muestra una secuencia de cuenta atrás de 4 segundos, enfocada a preparar a los jugadores de cara al inicio de la carrera. Cuando esta cuenta atrás llega al final, la barrera que impide a los jugadores empezar se elimina y da comienzo el juego (figura 16, abajo a la derecha).

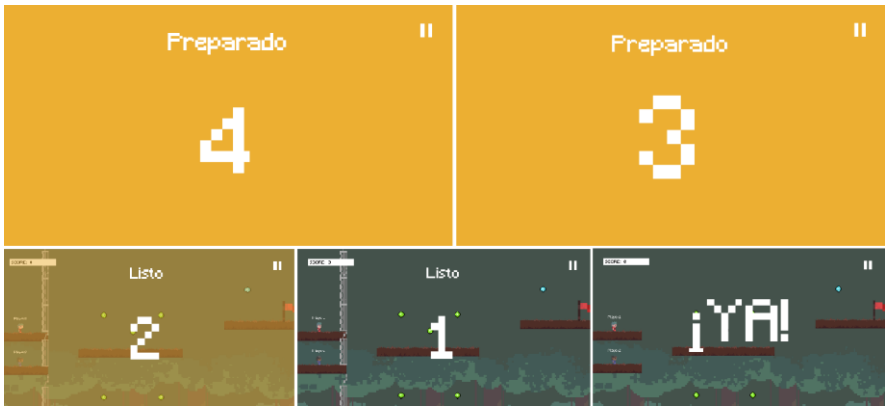


Figura 16. *Secuencia de cuenta atrás*

En el nivel de juego, la interfaz se reduce a un campo de puntuación, "*Score*", en la esquina superior izquierda (figura 17, arriba). Este campo contabiliza la puntuación que ha acumulado el jugador a lo largo del nivel gracias a los coleccionables recogidos (figura 17, abajo). Es visible para el jugador en cualquier punto de la carrera.

Figura 17. *Campo "Score"*

Al llegar al final del nivel, se muestra una pantalla con la posición en la que ha entrado el jugador junto a su puntuación total. Si el jugador que alcanza la meta no es el jugador *host*, verá el botón "Terminar" (figura 18, arriba), que, al pulsarlo, le devolverá a la pantalla de gestión de partidas. En el caso de que sea el jugador *host*, este botón estará desactivado hasta que no hayan llegado a la meta (o se hayan desconectado) el resto de los jugadores (figura 18, abajo). Cuando pulse, se deshará la partida, expulsando a los jugadores que todavía no hubieran pulsado el botón "Terminar".

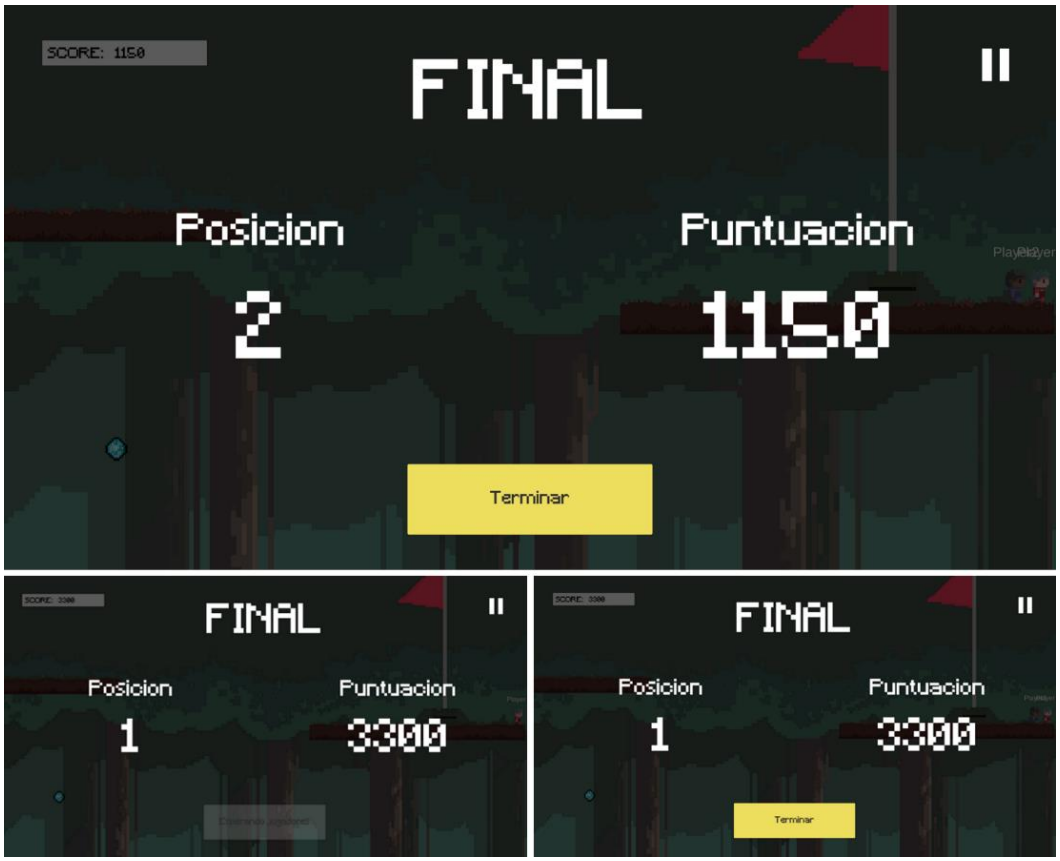


Figura 18. *Pantalla de final de nivel*

A lo largo del juego, tanto en la sala de espera (figura 19, izquierda) como durante el nivel de juego (figura 19, derecha), el usuario tendrá la posibilidad de desconectarse en cualquier momento. Para desplegar el menú correspondiente, podrá seleccionar el icono de pausa disponible arriba a la derecha o pulsar la tecla Esc. Una vez ahí, saldrá de la partida pulsando el botón "Desconexión".

Figura 19. *Menú en las diferentes escenas del juego*

También forman parte de la interfaz los mensajes de error. En este proyecto, se contemplan tres posibles errores:

- El usuario se conecta a una partida, local u *online*, pero mientras lo hace esta deja de estar disponible.
- El usuario se conecta a una IP incorrecta o que no está ejecutando una instancia del juego.
- El usuario se conecta a una partida local que ya está llena.

Cuando un usuario se conecta a una partida, lo primero que va a ver es un mensaje de espera con una animación (figura 20). Esta busca reducir la impaciencia del jugador. Al ver un elemento con movimiento será consciente de que se está llevando a cabo la conexión, mientras que en caso contrario podría llegar a pensar que el juego se ha bloqueado.

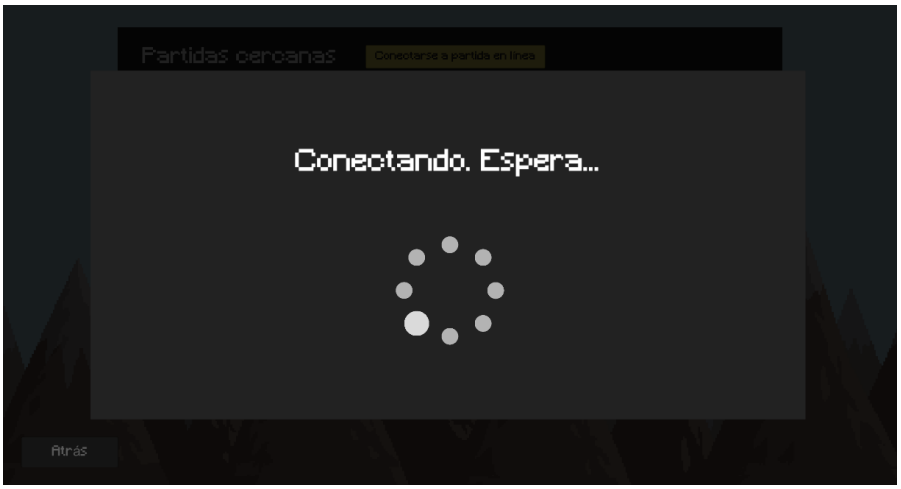


Figura 20. *Pantalla de espera*

Este mensaje dura unos 3 segundos. Mientras se muestra, internamente se está llevando a cabo un intento de conexión. En caso de que la conexión no haya sido posible porque el servidor ha dejado de estar disponible, se mostrará el correspondiente mensaje de error (figura 21).

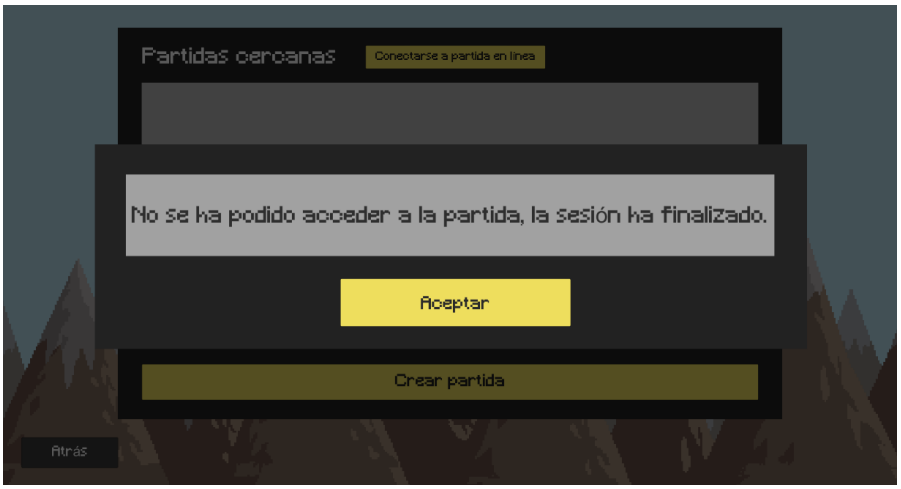


Figura 21. *Mensaje de error al conectarse a un servidor no disponible*

Si se introduce una IP incorrecta o que no corresponde a ningún servidor activo, se notificará el error al usuario mediante otro mensaje (figura 22).

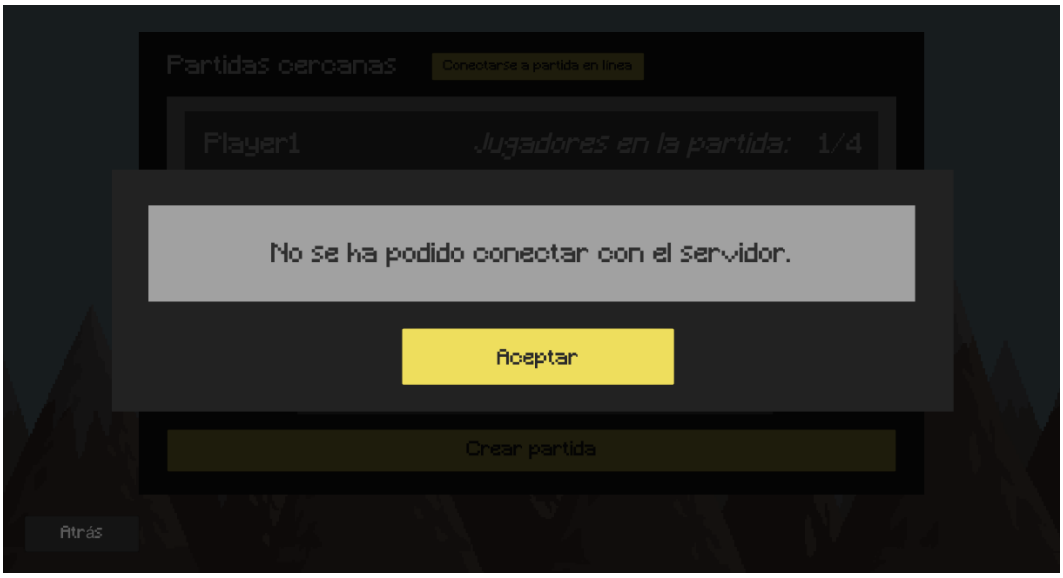


Figura 22. *Mensaje de error al introducir una dirección IP no válida*

Finalmente, el usuario podría intentar conectarse a una partida llena. Estas partidas se muestran en la lista de partidas cercanas, aunque no sea posible acceder a ellas. De esta forma, el usuario sabrá que la partida existe. Sin embargo, si decide unirse, verá automáticamente el conveniente mensaje de error (figura 23), sin pantalla de espera de por medio.

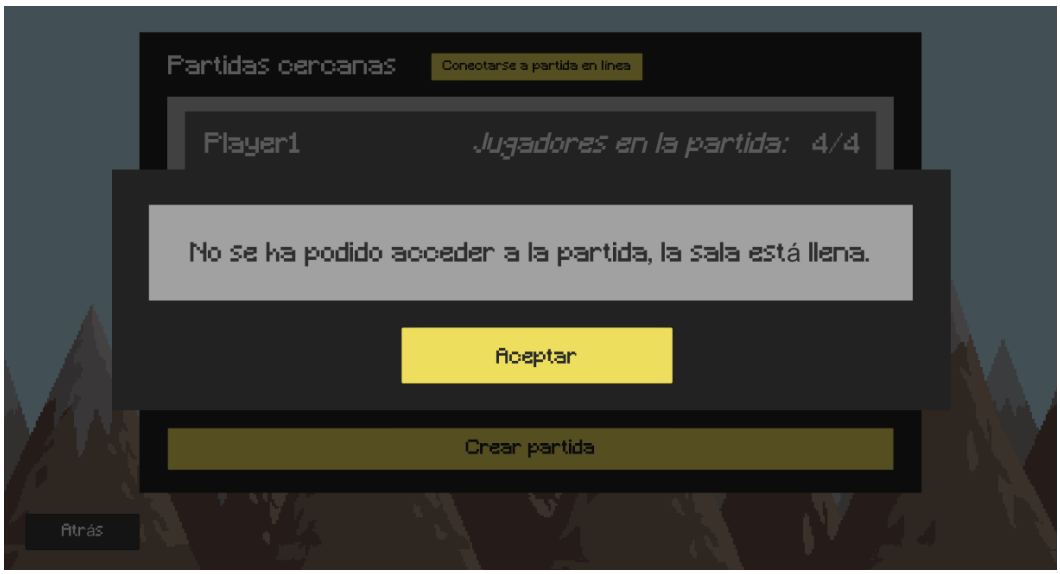


Figura 23. *Mensaje de error al unirse a una sala llena*

Estos son todos los elementos de la interfaz de usuario, que componen la capa de presentación. Como se puede ver, la interfaz es muy sencilla en cuanto a opciones. El esquema de color también es sencillo y de grandes contrastes que facilitan al usuario identificar los elementos de interés.

5.2 Capa de negocio

Dentro de la lógica que conforma el videojuego, se pueden distinguir dos ámbitos muy diferenciados: la que controla el juego y la que controla las funcionalidades multijugador. Teniendo en cuenta esta división, los *scripts* del proyecto se organizan de la siguiente manera:

- **Juego**
  - **Jugador**
    - *PlayerController.cs*
    - *PlayerScore.cs*
    - *PlayerCamera.cs*
  - **Escenario**
    - *SpawnController.cs*
    - *ScoreCollectible.cs*
    - *Checkpoint.cs*
    - *DeadZone.cs*
    - *FaceRight.cs*
    - *JumpingPlatform.cs*
    - *LimitPosition.cs*
    - *BackgroundScroll.cs*
  - **Interfaz**
    - *DiscoveryUIController.cs*
    - *LocalPlayerUI.cs*
    - *GetReadyUI.cs*
    - *MatchProperties.cs*
    - *MessageHandler.cs*
    - *ChangeScene.cs*
    - *QuitButton.cs*
- **Multijugador**
  - **NetworkManager**
    - *NetworkRoomManagerRunner.cs*
    - *NetworkRoomPlayerRunner.cs*
  - **NetworkDiscovery**
    - *NetworkDiscoveryCustom.cs*
    - *NetworkDiscoveryHUDCustom.cs*

Sin embargo, antes de detallar la lógica en sí, es importante describir cómo se distribuyen los elementos del juego. En *Unity*, los juegos se disponen en escenas. Una escena es un espacio que aloja objetos de juego (*GameObjects*), formados por componentes que definen su posición, apariencia y comportamiento, dispuestos de tal forma que construyan un nivel del juego. En este proyecto se han creado cuatro escenas: dos de menús y dos de juego.

- **01_Offline_1:** Esta es la escena que se muestra al arrancar el juego. Contiene los elementos de la interfaz que representan el logotipo, los fondos y los botones de "Jugar" y "Salir". El logotipo tiene definida una animación muy sencilla que le permite moverse hacia arriba y hacia abajo, como si estuviera flotando. Para las nubes se ha escrito un pequeño *script*, *BackgroundScroll.cs*, que crea un efecto de desplazamiento infinito sobre estas. El movimiento del logotipo, sumado al de las nubes, otorga algo de dinamismo a una escena que, de otra manera, sería muy estática. Finalmente, la lógica del botón "Jugar", definida por *ChangeScene.cs*, hace una transición a la siguiente escena; mientras que la del botón "Salir", en *QuitButton.cs*, cierra el juego.
- **01_Offline_2:** Aunque esta escena también es de menú, cuenta tanto con lógica de juego como de multijugador. La primera es muy sencilla, puesto que solo controla la interfaz de la escena a través de los *scripts DiscoveryUIController.cs* y *ChangeScene.cs*. La segunda está presente en *NetworkRoomManagerRunner.cs*, *NetworkDiscoveryCustom.cs* y *NetworkDiscoveryHUDCustom.cs*, pero más adelante se detallará su comportamiento.
- **02_Lobby:** Esta escena es la que representa la sala de espera a la que se unen los jugadores cuando crean o se conectan a una partida. Para marcar los límites de la escena, y así impedir el avance del jugador fuera de la pantalla, se utiliza el *script LimitPosition.cs*. El juego está diseñado para ejecutarse en diferentes plataformas, así que está preparado para redimensionarse en función del tamaño de la ventana o pantalla del dispositivo que lo ejecuta. Gracias a *LimitPosition.cs*, los límites de esta escena se ajustan a dicha pantalla. Por último, también estará presente en la escena la lógica de jugador y de multijugador cuando se instancie el avatar del usuario.
- **03_Game:** Esta escena es la que contiene el nivel del juego. Cuando los jugadores aparecen, se muestra la cuenta atrás, que está controlada por *GetReadyUI.cs*. Frente a ellos se disponen plataformas y paredes sobre las que podrán desplazarse, incluyendo unas plataformas que actúan como trampolín, manejadas por *JumpingPlatform.cs*. Mientras avanzan, encontrarán objetos coleccionables que, mediante el *script ScoreCollectible.cs*, incrementarán su puntuación. También pasarán por banderines controlados por *Checkpoints.cs*, que actúan como puntos en los que reaparecerán los jugadores después de caer al vacío. Este vacío incluye el *script DeadZone.cs*, que reduce la puntuación del jugador a la mitad y lo hace reaparecer transcurridos dos segundos en el último banderín por el que ha pasado.

5.2.1 Lógica del juego

La lógica del juego es aquella que controla el funcionamiento de los aspectos que se refieren a las mecánicas, esto es, el funcionamiento del jugador y los niveles. Los *scripts* agrupados bajo la categoría "Juego" en el esquema anterior entrarían dentro de esta lógica.

Controlador del jugador

El *script* más relevante de los que conforman la lógica del jugador es *PlayerController.cs*, que permite al usuario manejar a su avatar en el juego.

Una clase en *Unity* cuenta con algunos métodos importantes que se ejecutan automáticamente en función del estado del *GameObject* al que esté asociado el *script*. Los más habituales son *Awake*, *Start*, *Update*, *FixedUpdate* y *LateUpdate*. Los dos primeros se ejecutan cuando se instancia el *GameObject*, siendo *Awake* el que se lanza cuando el objeto se está cargando y *Start* el que lo hace justo antes del primer método *Update*. *Update*, *FixedUpdate* y *LateUpdate* son funciones dependientes de los fotogramas del juego. *Update* se ejecuta cada *frame*, independientemente de la diferencia de tiempo entre el *frame* actual y el anterior. *FixedUpdate* lo hace con una periodicidad fija, por defecto cada 0.02 segundos, por lo que existe la posibilidad de encontrar *frames* en los que esta función no se ejecute. Finalmente, *LateUpdate* se ejecuta después de *Update*, por lo que comparte su frecuencia.

Como *Update* se ejecuta constantemente, es la función encargada de recoger los *inputs* del jugador, mientras que se reflejan en *FixedUpdate*, ideal por su frecuencia fija.

```
78      private void Update()
79      {
80          if (!hasAuthority)
81              return;
82
83          // Se calcula la velocidad efectiva como el resultado de la velocidad base incrementada en la puntuacion del jugador, con cota superior una velocidad de 3
84          _effectiveSpeed = baseSpeed + (GetComponent<PlayerScores>().GetScore() * speedMultiplier);
85          _effectiveSpeed = _effectiveSpeed >= maxSpeed ? maxSpeed : _effectiveSpeed;
86
87          _isGrounded = Physics2D.OverlapBox(groundCheck.transform.position, new Vector2(0.2f, 0), 0, groundLayer);
88          _isGrabbingWall = Physics2D.OverlapCircle(wallCheck.transform.position, 0.025f, wallLayer) && !_isGrounded;
89
90          if (!_isActive)
91              return;
92
93          // Determina la direccion
94          float horizontal = 0;
95          float vertical = 0;
96
97          // Si esta en PC, se coge el valor de los ejes
98      #if UNITY_STANDALONE
99          horizontal = Input.GetAxisRaw("Horizontal");
100         vertical = Input.GetAxisRaw("Vertical");
101     #endif
102
103     #if UNITY_ANDROID
104         // Si esta en Android, se cojen los ejes del joystick y se mira si ha habido algun toque en la parte derecha de la pantalla
105         if (joystick.Horizontal >= 0.2f)
106             horizontal = 1;
107         else if (joystick.Horizontal <= -0.2f)
108             horizontal = -1;
109         else
110             horizontal = 0;
111
112         vertical = joystick.Vertical < -0.5f ? -1 : 0;
113
114         foreach (Touch touch in Input.touches)
115             if (touch.phase == TouchPhase.Began && !EventSystem.current.IsPointerOverGameObject(touch.fingerId) && touch.position.x >= Screen.width / 2)
116                 _jumpTouch = true;
117     #endif
118
119         // Determina el movimiento: Si esta agarrado a una pared, no se mueve
120         _movement = !_isGrabbingWall ? new Vector2(horizontal, 0f) : new Vector2(0, 0);
121
122         // Si toca el suelo o se agarra a la pared, se restaura el doble salto
123         if (_isGrounded || _isGrabbingWall)
124             _isDoubleJumping = false;
125
126         // Si estando agarrado a una pared y se pulsa hacia abajo, se gira y, por tanto, se suelta
127         if (_isGrabbingWall && vertical < 0)
128             Flip();
129
130         // Player Menu
131         if (Input.GetButtonDown("Menu"))
132             GetComponent<LocalPlayerUI>().SwitchPlayerMenu();
133
134         // Salto
135         if (Input.GetButtonDown("Jump") || _jumpTouch)
136         {
137             // Si esta en el suelo, aplica un impulso vertical
138             if (_isGrounded)
139                 _jump = true;
140
141             // Si esta en el aire y no ha realizado ya un doble salto, puede hacer un doble salto
142             if (!_isGrounded && !_isGrabbingWall && !_isDoubleJumping)
143             {
144                 _isDoubleJumping = true;
145                 _doubleJumpAnimator = true;
146                 _doubleJump = true;
147             }
148
149             // Si esta agarrado a una pared, indica que se ha saltado desde pared y gira al personaje para que salte en la direccion contraria
150             if (_isGrabbingWall)
151             {
152                 Flip();
153                 _wallJump = true;
154             }
155
156             _jumpTouch = false;
157         }
158
159         // Gira al personaje en funcion de la direccion. Si esta agarrando una pared, no se gira
160         if (horizontal < 0f && _facingRight && !_isGrabbingWall)
161             Flip();
162         else if (horizontal > 0f && !_facingRight && !_isGrabbingWall)
163             Flip();
164     }
```

***Código 1**. Método Update de PlayerController*

Como se puede ver en el código 1, *Update* inicializa las variables que determinan el movimiento del jugador: calcula valores como la dirección (variando el método en función de la plataforma en la que se ejecuta) y la posición, y comprueba si el jugador a pulsado algún botón. Dentro de la clase se utilizan diferentes variables globales booleanas para determinar el estado del jugador en todo momento. Así, en función de la dirección, la posición y los inputs del jugador se da valor a estas variables, que van a determinar qué acción va a llevar a cabo el jugador. Por ejemplo, si el usuario pulsa el botón de salto, solo se reflejará en el juego si el jugador está en el suelo o está en el aire y no ha efectuado ya un salto antes. Estas variables van a ser empleadas en *FixedUpdate* (código 2) para aplicar las acciones, deteniendo al jugador si está agarrado a una pared o imprimiendo fuerza si hay un salto. Finalmente, en *LateUpdate* (código 3) se hacen efectivos los cambios en lo que respecta a la animación del personaje, nuevamente usando estas variables globales.

```
169     private void FixedUpdate()
170     {
171         if (!hasAuthority)
172             return;
173 
174         // Si no esta agarrando una pared, se aplica un movimiento normal
175         // Si esta agarrando una pared, no se mueve en vertical (solo afecta la gravedad)
176         if (!_isGrabbingWall)
177             _playerRigidbody.velocity = new Vector2(_movement.x * _effectiveSpeed, _playerRigidbody.velocity.y);
178         else
179             _playerRigidbody.velocity = new Vector2(_movement.x * _effectiveSpeed, 0);
180 
181         // Si efectua un salto, se aplica la fuerza de jumpForce al Rigidbody2D
182         if (_jump)
183         {
184             _playerRigidbody.velocity = new Vector2(_movement.x * jumpForce, jumpForce);
185             _jump = false;
186         }
187 
188         // Si efectua un doble salto, se aplica la fuerza de doubleJumpForce al Rigidbody2D
189         if (_doubleJump)
190         {
191             _playerRigidbody.velocity = new Vector2(_movement.x * doubleJumpForce, doubleJumpForce);
192             _doubleJump = false;
193         }
194 
195         // Si efectua un salto de pared, aplica la fuerza de wallJumpForce al Rigidbody2D
196         if (_wallJump)
197         {
198             _playerRigidbody.velocity = new Vector2(_movement.x * wallJumpForce, wallJumpForce);
199             _wallJump = false;
200         }
201     }
```

Código 2. *Método FixedUpdate de PlayerController*

```
203     /// <summary>
204     /// Aplica los cambios sobre el componente Animator.
205     /// </summary>
206     private void LateUpdate()
207     {
208         if (!hasAuthority)
209             return;
210 
211         _animator.SetBool("Idle", _movement == Vector2.zero);
212         _animator.SetBool("Jumping", !_isGrounded);
213         _animator.SetBool("DoubleJump", _doubleJumpAnimator);
214         _animator.SetBool("TouchingWall", _isGrabbingWall);
215         _animator.SetFloat("VerticalVelocity", _playerRigidbody.velocity.y);
216 
217         _doubleJumpAnimator = false;
218     }
```

Código 3. *Método LateUpdate de PlayerController*

El resto de las funciones de *PlayerController* están enfocadas a apoyar el comportamiento del jugador, como una rutina para mover su *sprite* en función de la dirección hacia la que se mueve el avatar, u otra que registra el último banderín por el que ha pasado.

Gestión de las propiedades de la instancia del jugador

La otra clase más relevante de la parte del juego es *SpawnController.cs*. Este *script* controla cómo se instancian los jugadores en el escenario. Cuando la lógica multijugador detecta que se ha conectado un jugador y lo va a instanciar, obtiene de *SpawnController*

la posición y el *sprite* con el que debe aparecer. Para ello, se mantienen distintas listas, una para el identificador de red de los jugadores que están en la partida, dos para los puntos de aparición (para la sala de espera y para el nivel de juego) y otra para los *sprites* que se van a asociar a los jugadores. Todas estas listas están mapeadas entre sí, de tal forma que el índice del identificador de un jugador sirve para acceder a las otras listas y así obtener su posición y su *sprite*.

Así pues, cuando un usuario se conecta, se registra su identificador añadiéndolo a la primera lista (código 4), mientras que se sustituye su valor por -1 si se desconecta (código 5), dejando así un hueco libre en la sala. Con este índice se obtiene su posición de aparición (código 6) y su *sprite* (código 7).

Como ya se ha mencionado, existen dos listas de puntos de aparición: los de la sala de espera y los del nivel de juego. Mediante el primer parámetro del método *ChoosePlayerSpawnPoint* se indica a cuál de estas listas debe accederse. En *ChoosePlayerCharacter*, sin embargo, se prescinde de dicho parámetro porque la lista de *sprites* es independiente de la fase en la que se encuentra el juego.

```
45    public void RegisterNewPlayer(int netId)
46    {
47        int emptySpace = _players.IndexOf(-1);   // Si hay un hueco libre dejado por un jugador que se ha desconectado, es tomado por el nuevo jugador
48        if (emptySpace >= 0)
49        {
50            _players[emptySpace] = netId;
51            return;
52        }
53
54        _players.Add(netId);
55    }
```

Código 4. *Método RegisterNewPlayer de SpawnController*

```
62    public void RemovePlayer(int netId)
63    {
64        int index = _players.IndexOf(netId);
65        if(index > -1)
66            _players[index] = -1;
67    }
```

Código 5. *Método RemovePlayer de SpawnController*

```
75    public Transform ChoosePlayerSpawnPoint(bool isRoomPlayer, int netId)
76    {
77        int index = _players.IndexOf(netId);
78        Transform spawnPoint = isRoomPlayer ? _lobbySpawnpoints[index] : _gameSpawnpoints[index];
79
80        return spawnPoint;
81    }
82
```

Código 6. *Método ChoosePlayerSpawnPoint de PlayerController*

```
88    public GameObject ChoosePlayerCharacter(int netId)
89    {
90        int index = _players.IndexOf(netId);
91        GameObject character = playerCharacters[index];
92
93        return character;
94    }
```

Código 7. *Método ChoosePlayerCharacter de PlayerController*

Coleccionables

Mientras el jugador recorre el nivel podrá recoger objetos que aumentarán su puntuación, otorgando más o menos puntos en función de su color (figura 24).

Figura 24. *Coleccionables*

La lógica de estos coleccionables se ha repartido entre los *scripts ScoreCollectible.cs* y *PlayerScore.cs*. El primero (código 8), asociado al *GameObject* que representa al coleccionable, detecta cuándo un objeto de tipo *Player* entra en contacto con él a través de la función *OnTriggerEnter2D*. Cuando lo hace, lanza *GetCollectible*, que comprueba si el coleccionable está disponible y, si lo está, lo desactiva, incrementa la puntuación del jugador en la cantidad adecuada y lo destruye. Se controla el acceso al coleccionable con el atributo *_available* para evitar carreras críticas cuando dos jugadores lo intentan coger a la vez: el primero que llegue lo deshabilita, impidiendo que el otro pueda recogerlo también.

```
9   public class ScoreCollectible : NetworkBehaviour
10  {
11      public uint collectibleScore;
12
13      private bool _available = true;
14
15      [ServerCallback]
16      private void OnTriggerEnter2D(Collider2D collision)
17      {
18          if (collision.gameObject.CompareTag("Player"))
19              GetCollectible(collision.gameObject);
20      }
21
22      private void GetCollectible(GameObject player)
23      {
24          if (!_available)
25              return;
26
27          _available = false;
28          player.GetComponent<PlayerScore>().AddScore(collectibleScore);
29          NetworkServer.Destroy(gameObject);
30      }
31  }
```

Código 8. *Clase ScoreCollectible*

La función *AddScore*, invocada en *ScoreCollectible*, pertenece al segundo *script*, *PlayerScore*, asociado como componente al objeto del jugador y encargado de gestionar su puntuación, ofreciendo funciones para incrementarla, reducirla y actualizarla. La cantidad de puntos se almacena en el atributo *_playerScore*, y cada vez que cambia de valor se actualiza el campo de puntuación de la interfaz (código 9), buscándolo por el *tag ScoreField* especificado en el inspector de *Unity*.

```
47      public void UpdateScore(uint oldValue, uint newValue)
48      {
49          if (!hasAuthority)
50              return;
51
52          GameObject.FindWithTag("ScoreField").GetComponent<Text>().text = newValue.ToString();
53      }
```

Código 9. *Método UpdateScore de PlayerScore*

Otros *scripts* menores del juego controlan aspectos como la cámara del jugador o la interfaz de usuario, pero por su simpleza o irrelevancia dentro de la intención del proyecto no se detallarán.

5.2.2 Lógica de las funcionalidades multijugador

Como ya se ha comentado en el apartado de Tecnologías, *Mirror Networking* es el *plugin* encargado de gestionar las funcionalidades multijugador.

Mirror Networking es *server authoritative*, esto es, por defecto el servidor tiene autoridad sobre todos los objetos del juego, a excepción del objeto del jugador. El servidor ejecuta la misma instancia del juego que los clientes, y mediante llamadas especiales los clientes se comunican con él para interactuar con los elementos sobre los que no tienen autoridad. Por tanto, es el servidor el que gestiona la instanciación de los jugadores y cualquier otro *GameObject* en red.

Todo objeto que deba mostrarse en red para los clientes debe contar con el componente *NetworkIdentity*, ya proporcionado por *Mirror Networking*. Controla la identificación única del objeto dentro de la red, y el sistema lo usa para ser consciente de su existencia.

Comunicación entre cliente y servidor: Llamadas remotas

Para llevar a cabo acciones en la red, la *API* proporciona instrucciones llamadas *Remote Procedure Calls* (*RPCs*). Gracias a su uso, los clientes pueden mandar órdenes al servidor, y viceversa. A través de las *RPCs*, el código etiquetado se ejecutará en el servidor o el cliente:

- ***Command*:** Los bloques *Command* son enviados de los objetos del cliente a los objetos del servidor, de tal forma que el código pasa a ser ejecutado por el servidor. Como se puede ver en el código 10, la etiqueta *[Command]* se coloca antes de la definición del método, cuyo nombre debe comenzar por *Cmd*. El código de *CmdStartButtonClick* se ejecutará como servidor.
- ***ClientRpc*:** Los bloques *ClientRpc* son enviados de los objetos del servidor a los objetos del cliente, de tal forma que el código pasa a ser ejecutado por el cliente. La estructura es la misma que la vista en el código 10, sustituyendo *[Command]* por *[ClientRpc]* y *Cmd* por *Rpc*.
- ***TargetRpc*:** Los bloques *TargetRpc* son enviados de los objetos del servidor a objetos de un cliente específico. Si el primer parámetro recibido por la función es de tipo *NetworkConnection*, el tipo de dato que identifica a un cliente en la red, el código se ejecutará solo para ese cliente. Si, por el contrario, es de otro tipo, el código se ejecutará en el cliente propietario del objeto que tiene el *script* asociado. Nuevamente, el modo de etiquetado es igual que el mostrado en el código 10, sustituyendo *[Command]* por *[TargetRpc]* y *Cmd* por *Target*.

```
120     [Command]
121     private void CmdStartButtonClick()
122     {
123         NetworkRoomManagerRunner room = NetworkManager.singleton as NetworkRoomManagerRunner;
124         if (room)
125             room.StartGameRunner();
126     }
```

Código 10. *Método CmdStartButtonClick de NetworkRoomPlayerRunner*

Con el uso correcto de estas directivas se construye la comunicación entre cliente y servidor.

El gestor de red

El pilar de *Mirror Networking* es el gestor de red, *NetworkManager*. Este componente se encarga de gestionar prácticamente todos los aspectos básicos de la red, como la creación o destrucción de los objetos, el cambio entre escenas o el control del estado del juego.

NetworkManager es una versión básica, y se puede usar sin modificaciones para crear un juego muy sencillo. Sin embargo, para este proyecto se necesita extender su funcionalidad, para lo cual *NetworkManager* facilita métodos virtuales que se podrán redefinir en un componente personalizado. En concreto, esas necesidades pasan por implementar un *lobby* que mantenga a los jugadores a la espera de moverse a un nivel de juego. Para ello, *Mirror Networking* ya cuenta con la clase *NetworkRoomManager*, que extiende a *NetworkManager* con capacidades para controlar salas, gestionando el número de jugadores máximo, el estado de los jugadores y el cambio de escenas entre *lobby* y *game*. Nuevamente, ofrece funciones virtuales para personalizar su comportamiento. *NetworkRoomManagerRunner* es el *script* que se ha creado redefiniendo ciertos métodos de *NetworkRoomManager* para ajustarlo a los requisitos recogidos.

Figura 25*. Componente NetworkRoomManagerRunner*

En primer lugar, cabe destacar que *NetworkRoomManager* (y, por extensión, *NetworkRoomManagerRunner*) distingue entre tres tipos de escenas:

- ***Offline***: Es la primera escena que contiene el *GameObject* con el *script* *NetworkRoomManager* asociado. Durante esta escena, el juego no ha establecido ninguna conexión, pero es el momento en el que va a hacerlo, ya sea creando un servidor, una partida o uniéndose a una existente. Cabe destacar que la primera opción, crear un servidor, no se ha implementado en el juego porque no se ha considerado relevante, pues consiste en crear una instancia de la partida sin jugadores conectados.
  En este juego, la escena *offline* es *01_Offline_2*. Como al desconectarse de una partida se vuelve a la escena *offline*, se decidió separar la escena de menú en 2, diferenciando la pantalla con el logotipo de la pantalla de selección de partidas para evitar volver a la primera al desconectarse.
- ***Room***: Es la escena que representa la sala de espera. Cuando un jugador crea o se une a una partida, el juego se mueve a esta escena, llevando consigo el *NetworkRoomManager*. Aquí, el jugador podrá cambiar su estado a *Ready*, poniéndose a la espera de que todos los miembros hagan lo mismo y así empezar la partida (aunque este comportamiento se ha modificado ligeramente, como luego se describirá). En este proyecto, la escena *room* es *02_Lobby*.
- ***Game***: Es la escena en la que se desarrolla el juego propiamente dicho. Una vez todos los jugadores están listos, el juego se mueve a esta escena, y el *NetworkRoomManager* con él. En este proyecto, la escena *game* es *03_Game*.

Estas escenas se añaden al componente (figura 25) en las secciones “Scene Management” y “Room Settings”. La primera sección distingue solo la sala *offline* y la primera escena *online*, que en este caso sería el *lobby*. En la segunda sección se distingue entre la escena que actuará como *room* y la que lo hará como *game*.

Otros atributos relevantes de este componente son:

- ***Max Connections*:** El número máximo de conexiones permitidas y, por extensión, el número máximo de jugadores que pueden unirse al juego.
- ***Player Prefab*:** En *Unity*, un *prefab* es un *GameObject* reutilizable. Es muy útil para definir objetos con ciertas propiedades y crear instancias suyas a lo largo del juego tantas veces como sea necesario. En este caso, este *prefab* es el que utilizará el gestor de red como objeto del jugador por defecto. Aunque es imprescindible especificar uno, no lo es utilizarlo. Como en este juego cada jugador va a tener un aspecto diferente, se ha controlado la creación de los objetos de jugador a través del *SpawnController*, como se ha descrito en el aparatado anterior.
- ***Player Spawn Method*:** En este proyecto, tanto el objeto del jugador como su aparición se ha controlado mediante *SpawnController* para garantizar un control total. Sin embargo, *Mirror Networking* proporciona el componente *NetworkStartPosition* que, asociado a un *GameObject*, actúa como punto de aparición de los objetos de jugador. Si se hubiese utilizado este mecanismo, en

el *NetworkRoomManager* se podría haber especificado el método de aparición: *Round Robin*, para que cada jugador aparezca en la siguiente posición disponible; o aleatorio, para que cada jugador aparezca en cualquiera de las posiciones definidas.

- ***Show Room GUI*:** Indica si se muestra la interfaz de la sala. Más adelante se hablará de esto.
- ***Min Players*:** El número mínimo de jugadores necesarios para comenzar una partida.
- ***Room Player Prefab*:** Similar a *Player Prefab*, con la diferencia de que este será el objeto del jugador en la sala de espera, mientras que el otro corresponde al objeto en el nivel de juego. De igual forma, esto se ha gestionado a través de *SpawnController*, por lo que el valor de este atributo carece de relevancia.
- ***Spawn Controller*:** Este atributo es el único que se ha añadido en la clase redefinida *NetworkRoomManagerRunner*, y corresponde al *prefab* de *SpawnController* utilizado para la gestión de los jugadores en la sala y el juego.
- ***Registered Spawnable Prefabs*:** Lista de *prefabs* que van a instanciarse en la red en algún momento.

El gestor de red aparece en la escena *offline*, y se mueve de escena a medida que el juego avanza. Para acceder a él desde otros elementos de la escena, se utiliza el patrón *singleton*, ya implementado en la clase por defecto.

Este componente, por defecto, funciona junto al componente *NetworkManagerHUD*, que construye una interfaz sencilla para crear y unirse a partidas (figura 26). El componente *NetworkRoomManager* cuenta también con otra interfaz por defecto que gestiona la sala de espera (figura 27), y esta es la que se activa o desactiva con el campo *Show Room GUI* descrito más arriba.

Figura 26. *NetworkManagerHUD, interfaz por defecto de NetworkManager*

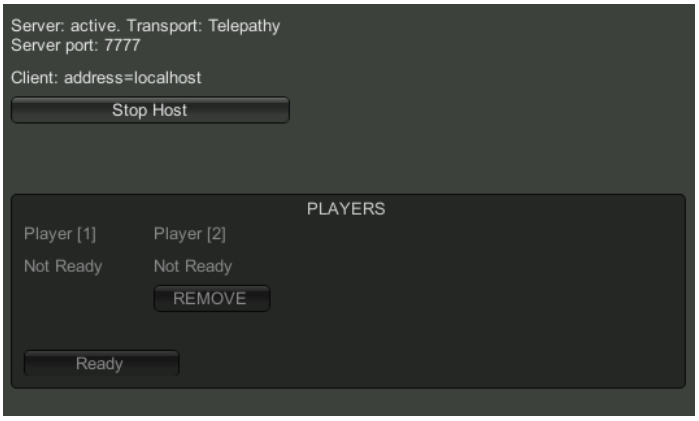


Figura 27. *Interfaz de sala de NetworkRoomManager* y *NetworkRoomPlayer*

Como se ha visto en el apartado de capa de presentación, ninguna de estas interfaces aparece. *Mirror Networking* las proporciona para tareas de depuración, pero realmente se espera que sean sustituidas por otras más elaboradas. En este proyecto, la interfaz de la figura 25 ha sido sustituida por la de la figura 8, eliminando la opción *Server Only*, mientras que la interfaz de la figura 26 se ha desactivado. Cabe destacar que la sección que muestra el nombre del jugador, así como un botón para cambiar su estado a Preparado, no pertenece a *NetworkRoomManager*, sino a *NetworkRoomPlayer*, cuyo comportamiento se describirá más tarde.

El resto de los métodos de *NetworkRoomManagerRunner* se ejecutan automáticamente como respuesta a eventos de la red. Proporcionan la lógica necesaria para intervenir el proceso de conexión de un cliente, cambio de escenas o de creación de un objeto de jugador en la sala (código 11), entre otros.

```
228     public override GameObject OnRoomServerCreateRoomPlayer(NetworkConnection conn)
229     {
230         bool isLeader = numPlayers == 0;
231         Transform spawnPoint = spawnController.ChoosePlayerSpawnPoint(true, conn.connectionId);
232         GameObject character = spawnController.ChoosePlayerCharacter(conn.connectionId);
233 
234         GameObject newRoomGameObject = Instantiate(character, new Vector3(spawnPoint.position.x, spawnPoint.position.y, -0.1f), spawnPoint.rotation);
235         newRoomGameObject.GetComponentInChildren<NetworkRoomPlayerRunner>().SetIsLeader(isLeader);
236 
237         return newRoomGameObject;
238     }
```

Código 11. *Función que interviene en la creación del jugador de sala*

El objeto del jugador

El primer componente de red importante en el jugador es *NetworkIdentity*, que como ya se ha explicado, es fundamental en los objetos que se comunican en la red.

El siguiente es *NetworkRoomPlayer*, otro componente que funciona de forma conjunta con *NetworkRoomManager*. Ha sido redefinido en *NetworkRoomPlayerRunner* para sustituir la interfaz de la figura 27 por la mostrada en las figuras 14 y 15, así como implementar la lógica necesaria para cambiar de estado y notificar al servidor.

```
92      private void ReadyButtonClick()
93      {
94          if (!hasAuthority)
95              return;
96 
97          Text buttonText = _readyButton.GetComponentInChildren<Text>();
98          ColorBlock buttonColors = _readyButton.GetComponentInChildren<Button>().colors;
99 
100         if (readyToBegin)
101         {
102             buttonText.text = "Listo";
103             buttonColors.normalColor = readyNormalColor;
104             buttonColors.selectedColor = readySelectedColor;
105             CmdChangeReadyState(false);
106         } else
107         {
108             buttonText.text = "Esperando";
109             buttonColors.normalColor = startNormalColor;
110             buttonColors.selectedColor = startSelectedColor;
111             CmdChangeReadyState(true);
112         }
113 
114         _readyButton.GetComponentInChildren<Button>().colors = buttonColors;
115     }
```

Código 12. *Método ReadyButtonClick de NetworkRoomPlayerRunner*

En la función *ReadyButtonClick* (código 12) se aplican cambios al botón para ajustar su apariencia en función de la situación. Pero lo realmente importante es la gestión del

estado, que usa la variable *readyToBegin* y la función *CmdChangeReadyState*, definidos en la clase por defecto *NetworkRoomPlayer* (código 13).

```
33          [SyncVar(hook = nameof(ReadyStateChanged))]
34          public bool readyToBegin;
70          #region Commands
71
72          [Command]
73          public void CmdChangeReadyState(bool readyState)
74          {
75              readyToBegin = readyState;
76              NetworkRoomManager room = NetworkManager.singleton as NetworkRoomManager;
77              if (room != null)
78              {
79                  room.ReadyStatusChanged();
80              }
81          }
99          public virtual void ReadyStateChanged(bool _, bool newReadyState)
00          {
01      #pragma warning disable CS0618 // Type or member is obsolete
02              OnClientReady(newReadyState);
03      #pragma warning restore CS0618 // Type or member is obsolete
04          }
```

Código 13. *Funciones ejecutadas al lanzar CmdChangeReadyState*

El uso de la etiqueta *SyncVar* en *readyToBegin* permite sincronizar la variable entre objetos de cliente y servidor, de tal forma que cuando se cambia su valor desde el servidor al ejecutar *CmdChangeReadyState*, el cambio se aplica también para los clientes. La función *ReadyStateChanged* se asocia como respuesta al cambio del valor de la variable (*hook = nameof(ReadyStateChanged)*), y simplemente ejecuta *OnClientReady*, una función virtual que se podría redefinir, aunque en este proyecto no se ha usado.

Finalmente, se añaden dos componentes que permiten sincronizar el movimiento en la red: *NetworkTransform* y *NetworkAnimator.* El primero, como su nombre indica, sincroniza el componente *Transform* del objeto, que controla su posición en la escena. El segundo sincroniza los cambios en su animación, consiguiendo que cada cliente vea el cambio en la animación de los demás objetos.

Descubriendo partidas locales

Para implementar la búsqueda de partidas locales se ha utilizado el componente *NetworkDiscovery*, que proporciona *Mirror Networking*. Este componente trabaja junto a *NetworkDiscoveryHUD*, la interfaz de usuario, para enviar paquetes a través de la red local, proporcionando información de la sala creada. Al igual que con el resto de los *scripts* de *Mirror Networking* vistos hasta ahora, estos proporcionan la funcionalidad básica, pero se pueden personalizar. Por defecto, solo se notifica la dirección del servidor y su identificador, así que se ha redefinido *NetworkDiscovery* en *NetworkDiscoveryCustom* para informar también sobre el nombre del jugador, el número de jugadores y si la sala es accesible o no (códigos 14 y 15).

```
public class DiscoveryResponse : MessageBase
{
    public IPEndPoint EndPoint { get; set; }
    public Uri uri;
    public long serverId;

    // Custom
    public bool accessible;
    public string hostPlayerName;
    public int totalPlayers;
}
```

Código 14. *Información enviada en el mensaje de notificación*

```
protected override DiscoveryResponse ProcessRequest(DiscoveryRequest request, IPEndPoint endpoint)
{
    try
    {
        DiscoveryResponse response = new DiscoveryResponse();
        response.serverId = ServerId;
        response.uri = transport.ServerUri();

        // Custom
        NetworkRoomManagerRunner room = NetworkManager.singleton as NetworkRoomManagerRunner;
        if (room)
        {
            response.accessible = room.AccesssibleRoom;
            response.hostPlayerName = room.roomSlots[0].GetComponent<LocalPlayerUI>().GetPlayerName();
            response.totalPlayers = room.roomSlots.Count;
        }

        return response;
    } catch (NotImplementedException)
    {
        Debug.LogError($"Transport {transport} does not support network discovery");
        throw;
    }
}
```

Código 15. *Construcción del mensaje de notificación*

```
private IEnumerator TryLocalConnection(GameObject selectedMatch)
{
    // Se toma la informacion de la partida a la que se intenta acceder
    Uri selectedMatchUri = selectedMatch.GetComponent<MatchProperties>().Uri;
    long selectedMatchId = selectedMatch.GetComponent<MatchProperties>().ServerId;

    GameObject waitMessage = ShowWaitMessage("Conectando. Espera...");

    // Refresca las partidas y da un tiempo para que se se anuncien los nuevos servidores
    RefreshMatches();
    yield return new WaitForSeconds(3f);

    // Se comprueba si el servidor al que se intenta conectar sigue listado
    bool hasBeenRemoved = true;
    foreach (GameObject match in listedServers.Values)
    {
        if (match.GetComponent<MatchProperties>().ServerId == selectedMatchId)
        {
            hasBeenRemoved = false;
            break;
        }
    }

    Destroy(waitMessage);

    // Si el servidor ya no esta listado, se muestra un mensaje y no se conecta
    if (hasBeenRemoved)
    {
        ShowUIMessage("No se ha podido acceder a la partida, la sesión ha finalizado.");
        yield return null;
    }

    NetworkManager.singleton.StartClient(selectedMatchUri);
    yield break;
}
```

Código 16. *Corutina TryLocalConnection de NetworkDiscoveryHUDCustom*

La interfaz, como es lógico, también se ha reemplazado. Redefinida en *NetworkDiscoveryHUDCustom*, cambia la interfaz de la figura 28 por la de la figura 9. También es en esta clase donde se ha implementado toda la lógica de auto refresco de las salas, que por defecto no se incluye, y los intentos de conexión y mensajes de error descritos en la capa de presentación. La función *TryLocalConnection* (código 16), por ejemplo, es la lanzada cuando se selecciona una partida de la lista.

Figura 28. *Interfaz por defecto de NetworkDiscoveryHUD*

Conexión a partidas online

Para implementar un mecanismo de descubrimiento de partidas *online*, *Mirror Networking* ofrece el componente *List Server*. Sin embargo, esta opción es de pago, así que no se ha añadido al proyecto.

La conexión a partidas *online* se lleva a cabo especificando la dirección IP del servidor al que unirse (figura 11). Este mecanismo no es el ideal porque va contra la premisa de juego rápido y directo. Sin embargo, al no contar con ningún servidor dedicado que gestionase el tráfico y crease instancias bajo demanda, se ha sustituido por el mecanismo más sencillo.

6. Pruebas

Para verificar el correcto funcionamiento del programa, se han llevado a cabo diferentes pruebas.

6.1 Pruebas unitarias

Las pruebas unitarias son aquellas diseñadas para verificar la funcionalidad aislada de una unidad de código. Para esto es necesario utilizar objetos que simulen el comportamiento de aquellos de los que depende la unidad bajo prueba, lo que ha resultado inviable en un proyecto de este tipo.

6.2 Pruebas de integración

Las pruebas de integración comprueban la correcta interacción entre los componentes del sistema. Solo los componentes presentes en una escena pueden interactuar entre sí, por lo que cada vez que se ha añadido un elemento nuevo al desarrollo se han comprobado sus posibles interacciones en las escenas en las que aparece. Haciéndolo de este modo, la integración ha sido incremental.

6.3 Pruebas de sistema

Las pruebas de sistema buscan garantizar el comportamiento del sistema completo, siendo especialmente adecuadas para comprobar los requisitos no funcionales.

6.3.1 Pruebas de portabilidad

El único requisito de este tipo consistía en poder ejecutar el juego en dispositivos *Android* y ordenadores de sobremesa (Windows, Mac y Linux), permitiendo el juego cruzado entre estas plataformas. Para garantizar este comportamiento, se ha ejecutado una instancia del juego en un *smartphone* con *Android* y otra en un PC con *Windows 10*. A continuación, se ha creado una partida en el *smartphone*, se ha unido a ella desde el PC y se ha podido completar una partida sin problema.

6.3.2 Pruebas de usabilidad

Respecto a la usabilidad, se plantearon dos requisitos.

El primero especificaba diseñar una interfaz accesible. Como prueba, se ha pedido a varias personas ajenas a los videojuegos que llevasen a cabo diferentes acciones en las diferentes plataformas. Tras las pruebas, ningún usuario ha tenido problemas para entender el funcionamiento del juego, por lo que se puede dar ese requisito por cumplido.

El segundo pedía un juego apto para todos los públicos. Para comprobar si esto se ha cumplido, se han utilizado las calificaciones presentes en el *Pan European Game Information* (*PEGI*), un organismo europeo que otorga calificaciones por edad a los videojuegos en base a su contenido (11). La etiqueta más permisiva es *PEGI 3*, que implica que el videojuego es recomendable para todos los grupos de edad (12). Para

optar a esta etiqueta el juego debe carecer de sonidos fuertes y violencia, por lo que cumple con los requisitos de *PEGI*.

Aunque se ha usado *PEGI* como referencia por ser el organismo que califica los juegos en España, otros como el *Entertainment Software Rating Board* (*ESRB*) en Estados Unidos, Canadá y México; o el *Computer Entertainment Rating Organization* (*CERO*) en Japón califican los juegos bajo criterios similares.

6.3.3 Pruebas de rendimiento

Tal y como se ha especificado en los requisitos no funcionales, los aspectos de rendimiento relevantes son los que se refieren a la memoria RAM y la cantidad de fotogramas por segundo.

Para determinar la cantidad de RAM consumida se ha hecho uso de la herramienta *Profiler*, integrada en el propio editor de *Unity*. Con esta utilidad se pueden monitorizar múltiples aspectos de consumo de recursos mientras se ejecuta el juego, entre ellos la memoria RAM. Así pues, se ha llevado a cabo una ejecución completa del juego, consiguiendo como máximo 1.58 GB de memoria RAM utilizada (figura 29).

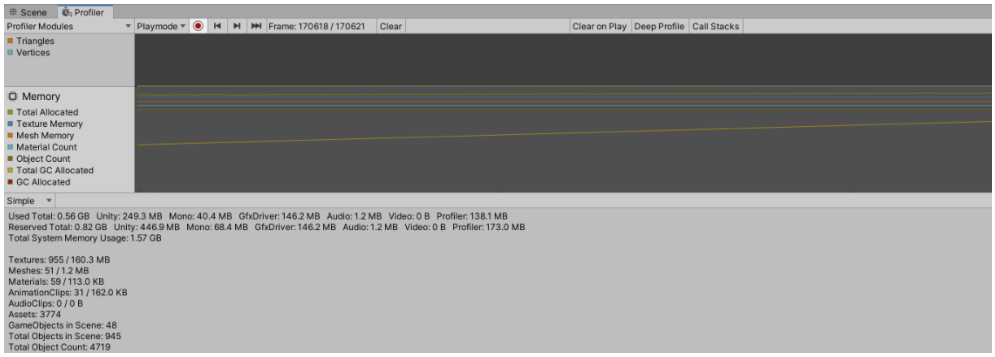


Figura 29. *Uso de la memoria según la herramienta Profiler*

Respecto a los fotogramas por segundo, se ha utilizado un pequeño *script* disponible en la wiki de *Unity* (13). Añadiéndolo como componente a un *GameObject*, se muestra arriba a la izquierda el contador de fotogramas por segundo (figura 30). Como se puede apreciar, este valor oscila en torno a los 140 FPS, por lo que el objetivo se cumple con creces.

Figura 30. *Estadísticas de rendimiento*

6.4 Pruebas de aceptación

Las pruebas de aceptación buscan que el cliente compruebe si todos sus requisitos han sido satisfechos. En este caso no hay cliente, pues los requisitos han sido auto impuestos. Así pues, superando todas las pruebas anteriores y cumpliéndose todos los requisitos definidos se puede dar el producto por aceptado.

7. Conclusiones y trabajos futuros

7.1 Conclusiones

A la vista de todo lo expuesto en el documento, considero que, durante este proyecto, he aplicado múltiples conocimientos que me serán de gran utilidad en mi futuro desarrollo como ingeniero informático. Por una parte, he comprobado de primera mano lo importante que resulta recoger correctamente los requisitos de un proyecto a su inicio, ya que esto permite acotarlo y facilita su planificación. Por otra, he tenido la oportunidad, por primera vez, de plantearme distintas metodologías y herramientas de desarrollo *software* y seleccionar las más adecuadas a la naturaleza y entorno del proyecto. *Unity3D*, por ejemplo, ha resultado ser una buena elección gracias a todo el material que he podido consultar *online* y que me ha ayudado a resolver los problemas que se me han ido planteando.

El haber aplicado estos conocimientos de ingeniería me ha permitido desarrollar un *software* de cierta complejidad, desarrollando desde cero un controlador de personaje y desplegando un sistema multijugador en tiempo real con gestión de salas.

Nunca había tenido contacto con el desarrollo de videojuegos, pero este proyecto me ha servido para ponerle remedio. Aunque siento más interés hacia el campo del diseño y la teoría de videojuegos, aplicar los conocimientos adquiridos durante estos años a un ámbito que tanta atracción me genera ha supuesto una introducción ideal. Gracias a ello he ganado otro punto de vista respecto al desarrollo, interesándome por aspectos que hasta ahora desconocía.

También me ha hecho darme cuenta de que realmente me gustaría intentar trabajar en esta industria. Durante este proyecto me he encontrado con múltiples baches que, en otra situación, no habría sido capaz de superar. Sin embargo, el gusto con el que he desarrollado este proyecto me ha permitido anteponerme y continuar hasta finalizarlo.

7.2 Trabajos futuros

Hay muchas cosas que habría querido incorporar a este videojuego, pero por razones de tiempo no ha sido posible. Cuando empecé a detallar el proyecto con mi director, surgieron ideas sobre diferentes sistemas para el juego, como objetos que otorgaran múltiples ventajas a los jugadores. Sin embargo, muchas de ellas se desecharon en favor de construir un sistema multijugador sólido, que por su complejidad requería de mucho más tiempo que otras partes del proyecto. Construir estas bases era muy importante porque, una vez hecho, todas las funcionalidades descartadas podrán ser añadidas sin mayor problema.

También habría querido experimentar implementando otras características interesantes, como tablas de clasificaciones o un sistema de *matchmaking* que permita jugar *online* sin la necesidad de especificar una dirección IP. Sin embargo, son ideas que han tenido que ser descartadas para no desviar los esfuerzos de los requisitos recogidos.

Con todo, va a resultar interesante seguir construyendo y mejorando el juego hasta crear un producto mucho más sólido y divertido.

Bibliografía

1. **Stewart, Samuel.** Video game industry silently taking over entertainment world. *Ejinsight.* [En línea] 2019. [Citado el: 12 de Agosto de 2020.] https://www.ejinsight.com/eji/article/id/2280405/20191022-video-game-industry-silently-taking-over-entertainment-world.

2. **TwitchTracker.** Most Watched Games on Twitch. *TwitchTracker.* [En línea] 2020. [Citado el: 12 de Agosto de 2020.] https://twitchtracker.com/games.

3. **Supercell.** Brawl Stars Championship. *Esports Brawl Stars.* [En línea] 2020. [Citado el: 12 de Agosto de 2020.] https://esports.brawlstars.com/es/.

4. **Chapman, Tom.** Fall Guys officially overtakes League of Legends as most-watched game. *GGRecon.* [En línea] 2020. [Citado el: 19 de Agosto de 2020.] https://www.ggrecon.com/articles/fall-guys-officially-overtakes-league-of-legends-as-most-watched-game?utm_source=twitter&utm_medium=other&utm_content=news#nnn.

5. **Unity 3D.** Real-time solutions, endless opportunities. *Unity.* [En línea] 2020. [Citado el: 15 de Agosto de 2020.] https://unity.com/solutions.

6. —. Plans and pricing. [En línea] 2020. [Citado el: 15 de Agosto de 2020.] https://store.unity.com/#plans-individual.

7. **Atlassian.** ¿Qué es kanban? *Atlassian.* [En línea] 2020. [Citado el: 15 de Agosto de 2020.] https://www.atlassian.com/es/agile/kanban.

8. **House, Brandy.** Evolving multiplayer games beyond UNet. *Unity Blog.* [En línea] 2018. [Citado el: 15 de Agosto de 2020.] https://blogs.unity3d.com/2018/08/02/evolving-multiplayer-games-beyond-unet/?_ga=2.172112736.2117709814.1598269876-1678090054.1595247230.

9. **Sommerville, Ian.** *Ingeniería del software.* 9ª edición. s.l. : Addison-Wesley, 2011. ISBN 978-6073206037.

10. **Vázquez Moreno, Juan Diego.** Introducción a Unity para videojuegos 2D. *Domestika.* [En línea] 2019. [Citado el: 16 de Agosto de 2020.] https://www.domestika.org/es/courses/716-introduccion-a-unity-para-videojuegos-2d.

11. **PEGI.** ¿Qué son las calificaciones? *PEGI.* [En línea] 2020. [Citado el: 26 de Agosto de 2020.] https://pegi.info/es/node/19.

12. —. ¿Qué significan las etiquetas? *PEGI.* [En línea] 2020. [Citado el: 26 de Agosto de 2020.] https://pegi.info/es/node/59.

13. **Unity Wiki.** FramesPerSecond. *Unity Wiki.* [En línea] 2017. [Citado el: 30 de agosto de 2020.]

http://wiki.unity3d.com/index.php?title=FramesPerSecond&_ga=2.224297043.918866032.1598726786-1678090054.1595247230.