
---

**Universidad de Valladolid**

## Escuela de Ingeniería Informática de Valladolid

### Trabajo de Fin de Grado

### Grado en Ingeniería Informática

### Mención en Ingeniería del Software

# Estudio y aplicación de un juego en red multijugador para dispositivos móviles

Presentado por:

***Carlos López Garcinuño***

Tutelado por:

***María Margarita Gonzalo Tasis***

Resumen

# Resumen

El ocio digital se presenta actualmente como una gran potencia económica, laboral y social. El crecimiento que ha tenido este sector y lo que ha aportado tanto a la economía como a la sociedad para de sorprendernos. Según AEVI [1], la Asociación Española de Videojuegos, el mercado global del videojuego creció un 8,5% en el año 2016 con respecto al año anterior alcanzando una facturación total de 99.600 millones de dólares (más de 92.000 millones de euros).

Teniendo en cuenta las posibilidades de este sector, más concretamente el mercado de los videojuegos en red, se presenta este TFG como representación del proceso de aplicación y aprendizaje de las herramientas necesarias para el desarrollo de un juego multijugador en red para dispositivos móviles, usando el motor de juegos Unity.

Resumen

# Abstract

Digital leisure is currently presented as a great economic, labor and social power. The growth that this sector has had and what it has contributed to both the economy and society do not stop surprising us. According to AEVI [1], the Spanish Association of Video Games, the global video game market grew by 8.5% in 2016 compared to the previous year, reaching a total turnover of $99.6 billion (more than 92 billion euros).

Taking into account the potential of this sector, more specifically the market of the network video games, this bachelor theses is presented as a representation of the application process and learning the necessary tools for the development of a networked multiplayer game for mobile devices, using the Unity game engine.

Abstract

# Índice de contenidos

# Índice de figuras

# Índice de tablas

Índice de tablas

# 1. Introducción

En este capítulo realizaremos una breve introducción del contenido de la memoria y el proyecto que ésta representa. Hablaremos sobre las motivaciones de la elección de dicho trabajo y los objetivos a cumplir.

## 1.1. Motivaciones

A lo largo de la historia reciente, la tecnología, con el afán de descubrir y de reinventarse, ha proporcionado a la humanidad diferentes y renovadas formas de realizar sus acciones cotidianas. Muchos de estos avances han sido orientados a la productividad, la organización y la eficiencia. Sin embargo, el ocio también ha sido uno de los aspectos más beneficiados por esta evolución.

Con el surgimiento de los videojuegos, el ocio para la gente tuvo un nuevo abanico de oportunidades abierto ante ellos. Seguían teniendo limitaciones, ya que para poder jugar con sus amigos tenían que seguir juntándose físicamente. Pero, al poco tiempo, la introducción de la red en los videojuegos proporcionó la posibilidad de jugar con amigos desde casa. La tecnología había acabado con esa limitación.

Considerando el potencial que poseen hoy día los videojuegos en red como herramienta para divertir y unir a la gente, hemos decidido enfocar este proyecto en la realización de un juego multijugador en red, para dispositivos móviles y poder explorar las posibilidades y resolver los problemas que éstos nos presentan.

## 1.2. Objetivos

En cuanto a los objetivos que queremos perseguir con la realización de este proyecto, hemos planteado los siguientes.

• Comprender el proceso de desarrollo de un juego desde cero.

• Adaptar los conocimientos adquiridos de Ingeniería del Software aprendidos en la carrera en el desarrollo del proyecto.

• Aplicar conceptos de gestión de proyectos y toma de decisiones.

• Aprender a utilizar de manera correcta todas las herramientas que nos brinda el motor del videojuego, así como el lenguaje de programación C#.

• Observar y analizar videojuegos similares al planteado con el objetivo de ayudar a resolver, como hacen ellos, ciertos problemas relacionados con la plataforma y con el tipo de juego en concreto.

• Aprender y aplicar el funcionamiento de la red en los videojuegos, consiguiendo así un videojuego con una buena jugabilidad para varios usuarios simultáneos.

• Aprender la gestión dentro del juego de los elementos que caracterizan a los juegos tipo MOBA y sus mecánicas.

• Estudiar cómo permitir jugar a los usuarios juntos en cada partida y construir una solución.

## 1.3. Organización de la memoria

A lo largo de este documento abordaremos todos los aspectos relativos al proyecto. Con el objetivo de situar al lector, se procede a hacer una breve descripción del contenido de cada capítulo.

• **Capítulo 1: Introducción**. Se presentan los objetivos y motivaciones del proyecto.

• **Capítulo 2: Contexto**. Se sitúa el proyecto en el contexto histórico, técnico y teórico.

• **Capítulo 3: Documento de Diseño del Juego**. Se describen las principales características específicas del videojuego.

• **Capítulo 4: Desarrollo del Proyecto**. Se presenta la metodología de desarrollo que se seguirá, así como la gestión de riesgos y la planificación.

• **Capítulo 5: Análisis**. Se detallan los requisitos y casos de uso del sistema.

• **Capítulo 6: Diseño**. Se describen y presentan los modelos de dominio pertinentes. Diagramas y patrones relativos al diseño de la aplicación.

• **Capítulo 7: Implementación**. Se explican los elementos relevantes de la implementación de la aplicación.

• **Capítulo 8: Pruebas**. Se presentan las pruebas a las que ha sido sometido el software y el resultado de ellas.

• **Capítulo 9: Manuales**. Se muestran los manuales de instalación y uso de la aplicación.

• **Capítulo 10: Conclusiones**. Valoraciones finales del proyecto, evaluación de la consecución de los objetivos marcados y posibles mejoras para el futuro de la aplicación.

# 2. Contexto

## 2.1. Breve historia de la red

Hace algo más de 20 años, las computadoras eran una novedad para la mayoría. Proporcionaban, al mismo tiempo, el mismo servicio que una máquina de escribir, una calculadora y un centro de entretenimiento, todo en un solo aparato. Las redes de computadoras eran algo completamente extraño para el usuario medio. La principal forma de compartir información de una máquina a otra era el disquete.

El gobierno de los Estados Unidos se dio cuenta de la importancia y potencial de la tecnología de internet y promovió una investigación que hizo posible un internet global. La tecnología DARPA incluía un conjunto de estándares de red que especificaban en detalle cómo los ordenadores debían comunicarse, así como convenciones para interconectar redes y el reenvío de tráfico. Oficialmente se llamó *TCP/IP Internet Protocol Suite* (Conjunto de protocolos de Internet) pero al final se quedó con TCP/IP [2] [3] [4], que son el nombre de sus dos estándares más principales e importantes. TCP significa Protocolo de control de transmisión, e IP, Protocolo de internet.

Pasó a ser el estándar a nivel mundial para la interconexión de sistemas. No hay ningún otro protocolo que ofrezca tanta interoperabilidad ni se ejecuta sobre tantas tecnologías de red como TCP/IP.

Uno de los protocolos que más nos va a interesar en las cuestiones que vamos a tratar, por la importancia que tiene en un campo como es el de los videojuegos, es el UDP. Este se utilizaba incluso antes del nacimiento de TCP, pero debido a la necesidad de la fiabilidad que UDP no podía garantizar, TCP tomó mucha importancia.

El Protocolo de Datagramas de Usuario [5] [6], o UDP (*User Datagram Protocol*) es un protocolo del nivel de transporte (al igual que TCP). La diferencia fundamental entre UDP y TCP es que UDP no proporciona obligatoriamente transmisiones de datos fiables. Es más, el protocolo no garantiza que los paquetes finalmente lleguen a su destino, como sí hace TCP. Aunque pueda parecer extraño, esto es útil para una serie de tareas. Tareas que requieren transmitir la máxima información en el menor tiempo posible, sin problemas de fiabilidad.

Por ejemplo, en las aplicaciones de *streaming* de video en directo, es más importante que la imagen y el sonido lleguen sin mucho retardo a que se vea totalmente completa y a máxima resolución (llegarían el 100% de los paquetes). También es el caso de los videojuegos multijugador online en tiempo real. Es más importante y necesario para una buena experiencia de juego que las acciones de los usuarios lleguen lo antes posible al servidor, a que lleguen todos y cada uno de los paquetes.

Aunque la gran mayoría de servicios van a necesitar el protocolo TCP porque realmente sí necesitan que todos y cada uno de los paquetes lleguen a su destino, la función que cumple UDP es algo que TCP no puede igualar. Muchos programas utilizan conexiones separadas TCP y UDP. La información importante se envía a través de una conexión TCP fiable, mientras que el flujo principal de datos se envía por UDP.

La explicación de por qué UDP es más rápido que TCP es la siguiente: el cometido de TCP es proporcionar transmisiones de datos fiables y mantener una conexión entre dispositivos o servicios que estén “manteniendo una conversación”. De este modo, si se recibe algún paquete corrupto, no se recibe, o llega en un orden incorrecto, TCP es el encargado de la recuperación de los datos. Para ello, utiliza una confirmación (ACK) de la máquina de destino, que es enviada una vez se hayan recibido correctamente los datos. UDP, al evitar todo esto, es claramente más rápido y eficiente para los casos mencionados anteriormente.

Otro motivo por el que es más rápido es que el paquete es mucho más ligero. Aquí tenemos el paquete de TCP.

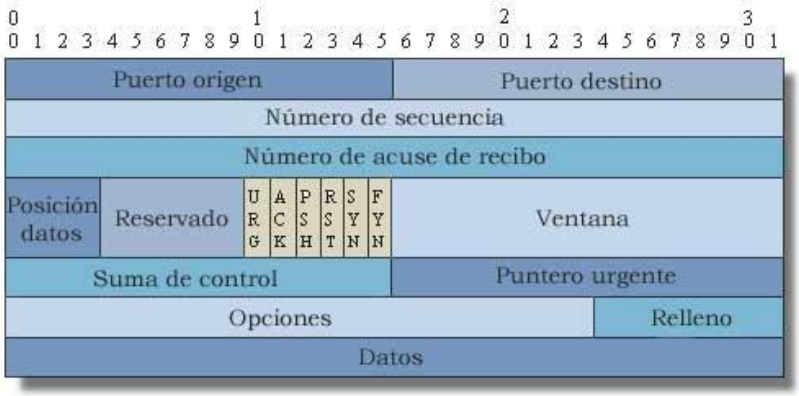


*Figura 2.1. Cabecera TCP Fuente: [7]*

Y aquí el de UDP, con muchos menos bits ocupados. Lo que le permite ser más pequeño, sin impedirle enviar la información realmente importante.

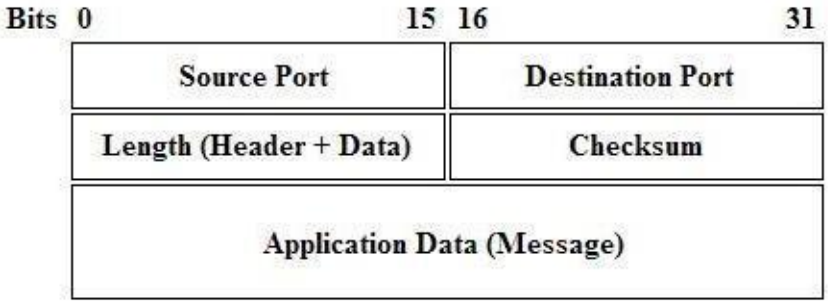


*Figura 2.2. Cabecera UDP Fuente: [8]*

Gracias a estos protocolos, la situación fue cambiando y las redes de computadores empezaron a llenar las aulas y oficinas. El correo electrónico nos permitió comunicarnos de forma global de manera instantánea. Ya no era necesario desplazar la información manualmente de un sitio a otro. Simplemente bastaba con conectarse con un módem y una línea telefónica y hacer las transacciones. Al dotar a las máquinas de capacidad para comunicarse entre sí, se permitió que la información fluyera, así nació la red de computadoras.

Tras esto, internet no hizo más que evolucionar, las líneas telefónicas y los módems dieron paso a las líneas de ADSL, fibra óptica e incluso se incorporó a los teléfonos móviles.

Actualmente estamos en esta fase de la tecnología, los *smartphones* son uno de los principales focos en los que se centra la innovación, y como éstos, las aplicaciones que pueden ejecutar.

## 2.2. Aplicaciones móviles en red

Las aplicaciones móviles ya forman parte de las herramientas de uso diario de la población. Se ha decidido centrar este trabajo en las aplicaciones en red, por el valor añadido que otorgan a las *offline*.

En los primeros *smartphones* la gran mayoría de las aplicaciones no hacían uso de internet, debido a su dependencia de una escasa red de datos móviles que la población no necesitaba en ese momento. Gracias, principalmente, a las aplicaciones de mensajería instantánea y a las redes sociales, esta necesidad aumentó considerablemente. Esto, junto con la evolución de las tecnologías y el consiguiente abaratamiento de los costes permitieron una democratización de los *smartphones*.

Esta nueva coyuntura impulsó un nuevo mercado: las aplicaciones móviles. Comenzó a ser más popular obtener tarifas de datos móviles, las aplicaciones *offline* comenzaron a ser insuficientes, lo que llevó al surgimiento de las aplicaciones *online*.

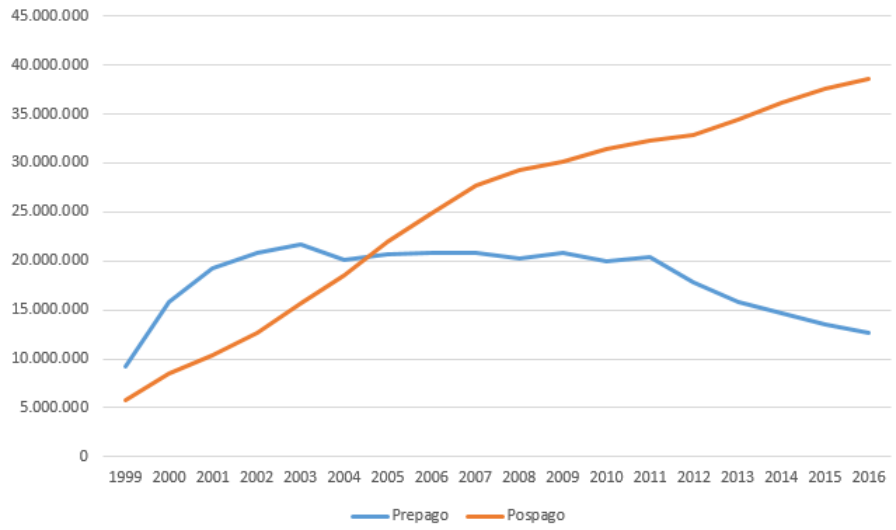


*Figura 2.3. Gráfico de la evolución de las líneas pospago y prepago*

Como podemos ver en la gráfica anterior, perteneciente a un artículo publicado en 2017 [9] a partir de 2011, que es aproximadamente donde empezó el *boom*, hay una bajada bastante pronunciada de las tarifas prepago. En cambio, las líneas pospago, conocidas también como “contrato”, no paran de crecer y han seguido en esa tendencia hasta ahora. Es extraño encontrar a alguien

actualmente que no disponga de una tarifa de llamadas y datos móviles con su smartphone. Esto es entendible ya que como mencionábamos anteriormente, hubo una democratización de los smartphones y no sólo de ellos, sino de estas tarifas pospago. Por ejemplo, en este artículo [10], de donde proviene la figura de abajo, podemos ver cómo, prácticamente por el mismo precio o incluso menor, los “megas” o “gigas” que se ofrecen son bastante superiores en las compañías principales de España.

| | 2010 | 2011 | 2012 | 2013 |
|---|---|---|---|---|
| Orange | Delfín 20: 100 MB + 300 min (de 18h a 8h) por 23,6€/mes | Delfín 20: 200 MB + 300 min (de 18h a 8h) por 23,6€/mes | Ballena 22: 1 GB + 100 min por 26,2€/mes | Delfín 16: 1 GB + 150 min por 19,36€/mes |
| Movistar | Internet en el móvil Premium: 1 GB por 29,5€/mes (llamadas y SMS a parte) | Internet 25 smartphone: 2 GB por 29,5€/mes (llamadas a parte) | Internet 20: 1 GB por 24,2€/mes (llamadas a parte) | Movistar 20: 1 GB + 250 min por 20€/mes |
| Vodafone | @XS: 100MB + 350 minutos (por las tardes y fines de semana) por 29,38€/mes | @XS: 100MB + 150 minutos por 23,6€/mes | @XS: 100MB + 150 minutos por 24,2€/mes | Smart S: 600 MB + 200 min por 18€/mes |

*Figura 2.4. Evolución de las ofertas de líneas pospago*

Al ir creciendo tanto la capacidad de procesamiento como el uso de los smartphones, internet se adaptó a ellos haciendo que las páginas web se pudieran leer perfectamente en dispositivos móviles. Si se hiciera una comparativa sobre la navegación desde el móvil en 2010 y en la actualidad, se mostrarían claramente los progresos existentes.

Así ha llegado hasta nuestros días, que como podemos comprobar [11] los móviles son un elemento indispensable de nuestra vida. Estos son algunos de los datos que lo reflejan: un 61% de los usuarios de móvil en el mundo miran su móvil en los primeros 5 minutos tras despertarse, la media de uso del móvil diaria son 170 minutos, o que el 99% de los jóvenes españoles accede a diario a internet desde su móvil. Una realidad que refleja que los móviles han llegado para quedarse, y las aplicaciones online, con ellos.

## 2.3. Videojuegos

Uno de los tipos de aplicaciones móviles online más usados son los videojuegos. Vamos a introducirnos un poco en este campo.

En 1972 [12] se desarrolla el primer juego, llamado PONG, que consistía en una especie de partida de ping-pong. En 1977, la empresa Atari lanzó al mercado el primer sistema de videojuegos en cartucho (Atari VCS/2600), que triunfó en los Estados Unidos. Tras una gran evolución, en la que la potencia de los procesadores era cada vez mayor, así como la memoria, favorecieron nuevas mejoras. En 1985, Nintendo lanzó su primera plataforma de videojuegos, la NES. Esta consola permitió la presentación de unos videojuegos que se creían imposibles años atrás.


*Figura 2.5. Nintendo NES*

La calidad de éstos que permitían los mejorados componentes, unido al ingenio de los creadores hizo que los videojuegos se popularizaran tanto durante la década de los 90 que llegaron a ser uno de los juguetes favoritos de los niños. La dinámica de innovación y mejora ha seguido en aumento hasta nuestros días, y así es como hoy en día, tenemos videojuegos que en ocasiones cuesta diferenciar si lo son en realidad o si se trata de una escena grabada con actores.

Dentro de los videojuegos hay muchas maneras de clasificarlos, una de ellas sería el número de jugadores que estos admiten. Diferenciamos así entre juegos para un solo jugador o juegos multijugador. Nos centraremos en los juegos multijugador, ya que serán el objetivo de este trabajo.

### 2.3.1. Videojuegos multijugador

Los comienzos de los juegos multijugador se remontan hasta el inicio mismo de los videojuegos. La razón primordial de esto, además de por el entretenimiento de más de una persona, se relacionaba también por la dificultad de diseñar un oponente "digno" para el jugador con los medios de los que se disponían en ese momento. Por ejemplo, el PONG. En ese momento era más fácil que un segundo oponente manejara, en el mismo dispositivo, la segunda "pala" a desarrollar un rival que pudiera contestar a los restos del jugador. Se conoce que los primeros juegos multijugador en tiempo real se desarrollaron en el sistema PLATO (Programmed Logic Automated Teaching Operations) [13], alrededor de 1973. En marzo de 1974 fue lanzado Spasim (Space Simulator) [14], está considerado como el primer videojuego de disparos en primera persona (FPS – First Person Shooter) multijugador en red.

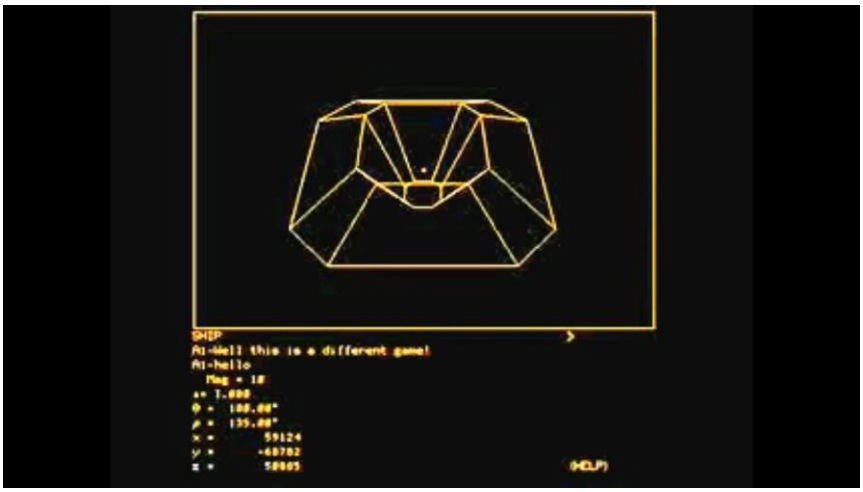

*Figura 2.6. Imagen de una partida de Spasim Fuente: [15]*

En los noventa, con el auge de internet, se expande el multijugador. Los oponentes no se limitaban a tu red local, sino que se extendían a todo el mundo. Algunos de los primeros juegos en ofrecer multijugador fueron Quake, Duke Nukem o Doom 2. A partir de ahí, las empresas de videojuegos estudiaron las posibles opciones para reenfocar su estrategia: o bien a un juego de un solo jugador le añadían una modalidad *multiplayer* o, por el contrario, se animaban a desarrollar un juego multijugador desde cero.

Ejemplo del primer grupo podríamos encontrarnos Half-Life (1998) [16] desarrollado por Valve, un juego cuyo principal interés era el modo historia de un solo jugador pero que tuvo un modo multijugador aprovechando las mecánicas ya desarrolladas para el modo historia. En cuanto al segundo grupo tenemos un

ejemplo, que viene del mismo desarrollador que Half-Life (de hecho, está desarrollado a partir de éste).

*Figura 2.7. Imagen durante una partida de Counter-Strike Fuente: [17]*

Estamos hablando de Counter Strike (1999) [18] considerado uno de los FPS más importantes de la historia. Este juego carece de un modo historia. Consiste en una serie de distintos mapas (desiertos, bases militares, rascacielos…) en la cual se enfrenta un grupo de jugadores (terroristas) contra otro (antiterroristas). La mecánica básica es intentar acabar con el equipo enemigo, o, por el contrario, existe otra manera de ganar, ya sea colocando una bomba (terroristas) o rescatando a los rehenes (antiterroristas).

Después de estos juegos, la situación ha seguido evolucionando, dando grandes títulos que han dado horas y horas de disfrute en línea a multitud de jugadores. Son juegos como, por ejemplo, StarCraft, World of Warcraft, Battlefield, Call Of Duty, Dota, League of Legends, etc. Todos estos juegos pertenecen, dentro de los juegos multijugador en línea, a distintas categorías, que entraremos a desarrollar a continuación.

## 2.3.2. Categorías de los videojuegos multijugador

### 2.3.2.1. Acción

Dentro del género de acción, tenemos varias categorías (arcade, plataformas, beat 'em up…), aunque, realmente, estas mencionadas prácticamente no tienen juegos multijugador relevantes. El género de luchas, en cambio, sí lo tiene. Juegos como Mortal Kombat y Super Smash Bros son muy jugados de manera competitiva por todo el mundo, lo que les dota de mayor relevancia.


*Figura 2.8. Smash Bros Ultimate Fuente: [19]*

### 2.3.2.2. Disparos

Conocidos también como *shooters,* podríamos incluirlos dentro del anterior punto, han pasado a ser una subcategoría tan grande y exitosa que se ha ganado un puesto como categoría propia. Hablamos de los juegos de disparos multijugador. Distinguiremos principalmente entre dos tipos.

#### 2.3.2.2.1. Disparos en primera persona (FPS – First Person Shooter)

Como nombramos anteriormente, uno de los más conocidos en este ámbito sería el Counter Strike. Pero tenemos grandes títulos como Battlefield o Call Of Duty. Habitualmente siguen dinámicas muy similares que consisten o bien en comprar distintas armas e intentar acabar con el equipo enemigo, o bien ejecutar distintos objetivos.

Otro juego muy popular considerado un FPS es el Overwatch, aunque tiene características que distan de los nombrados anteriormente y ofrece mecánicas distintas para cada uno de los personajes que conforma el juego.

#### 2.3.2.2.2. Disparos en tercera persona (TPS – Third Person Shooter)

Muy similares en cuanto a mecánicas de los FPS pero con la diferencia de que la cámara se encuentra en tercera persona. Suele ser útil para juegos en los que el objetivo no es solo disparar, sino que, por ejemplo, el jugador debe recoger objetos del suelo y por ello es necesario tener más conocimiento espacial.


*Figura 2.9. Jugador recogiendo objetos del suelo en Fortnite Fuente: [20]*

Dos juegos muy populares en esta categoría que realmente han tenido un gran éxito en estos últimos años son PlayerUnknown's Battlegrounds (PUBG) y Fortnite, ambos con versión para smartphone.

### 2.3.2.3. Estrategia

Este tipo de juegos requieren que el jugador ponga en práctica sus habilidades de planeamiento e inteligencia para maniobrar, gestionando recursos de diverso tipo (materiales, humanos, militares...) para conseguir la victoria. La mayoría son de temática bélica, pero los hay también centrados en estrategia económica o empresarial.

#### 2.3.2.3.1. Estrategia en tiempo real (RTS – Real Time Stategy)

Dentro de la estrategia en tiempo real, podemos distinguir dos.

El primero lo conformarían juegos como el Imperivm, Warcraft, StarCraft, etc. La principal diferencia con el segundo grupo es que éstos se juegan en una serie

de mapas, para una partida en concreto. Y una vez acabada la partida ese mapa desaparece, como pasa con los mapas mencionados en Counter Strike. Son mapas de “usar y tirar”, las partidas pueden durar horas, pero no mucho más.

*Figura 2.10. Imperivm Fuente: [21]*

En el segundo grupo (MMORTS) tendríamos juegos como OGame y Clash Of Clans. Como decíamos, a diferencia del primer grupo, el mapa no es de “usar y tirar”, sino que en éste, las unidades, las estructuras y los recursos, se mantienen, en un mundo persistente que evoluciona independientemente de que los jugadores estén conectados o no. Este tipo de juegos triunfaron mucho como juegos de navegador.

#### 2.3.2.3.2. Estrategia por turnos (TBS – Turn Based Stretegy)

Este tipo de juegos, entre los que se encuentran los Civilization y la saga Total War, se caracterizan por dividir el flujo del juego en varias partes bien definidas llamadas turnos o rondas. En el caso de estos dos juegos mencionados, el juego se suele desarrollar en un mapa del territorio en el que vas organizando tus ejércitos, planteando políticas y atacando o defendiendo otros territorios. Al final de tu turno, el rival (o rivales) realiza las mismas acciones hasta que te vuelve a ceder el turno. Estos juegos a menudo incorporan la posibilidad de manejar ciertas batallas como si de un RTS se tratara.


*Figura 2.11. Medieval II - Total War Fuente: [22]*

### 2.3.2.4. Deporte

Esta categoría se centra en videojuegos sacados de deportes reales llevados al terreno virtual. Se encuentran juegos como el FIFA, el Pro Evolution Soccer, NBA. Todos estos juegos tienen una gran *fanbase* y mucha gente que juega al modo multijugador, sobre todo, destaca el FIFA, que posee un modo multijugador online en el que puedes crear tu equipo de fútbol y competir en una liga con jugadores de todo el mundo.

### 2.3.2.5. Carreras

En cuanto a los videojuegos de carreras, mucha similitud con la sección anterior. Carreras, generalmente de coches, simuladas en el juego. Estas simulaciones pueden ir desde las más realistas, como por ejemplo Fórmula 1 o Gran Turismo, pasando por algo más fantástico como Need For Speed, hasta llegar a Mario Kart, inspirado en el universo de Nintendo.

### 2.3.2.6. Aventura

Son videojuegos habitualmente basados en una trama en la que el usuario va avanzando, interactuando con personajes y resolviendo las situaciones que van apareciendo. Este género dedica la mayoría de sus títulos a un solo jugador, dando mucha importancia al modo historia, como pueden ser The Last Of Us, Uncharted y God Of War.

Sin embargo, hay alguno que sí incorpora algún tipo de modo multijugador, como puede ser Left 4 Dead y Assassins Creed.

### 2.3.2.7. Rol

Aunque pueden enmarcarse en la sección de Aventuras, el género de juegos de rol (RPG) tiene tantos títulos y éxito que ha pasado a ser un género propio, como ocurre con los *Shooter*. Tienen todo lo que tiene un juego de aventuras (personajes y trama) pero hay un desarrollo del personaje en el que va adquiriendo armas, habilidades nuevas o aliados. Ejemplos de estos juegos podrían ser Skyrim, Dungeons & Dragons o incluso Pokémon. Los juegos nombrados no son online, o, al menos, en su primer diseño.

Para ir al terreno online, tenemos que hablar de los MMORPG (*massively multiplayer online role-playing game).* Como su nombre indica, se trata de juegos multijugador online masivos, lo que significa que pueden albergar a miles y miles de jugadores de forma simultánea en un mundo permanente, que nunca duerme.


*Figura 2.12. Expansiones de las que dispone World of Warcraft Fuente: [23]*

El título más conocido de esta categoría es el World Of Warcraft [24], lanzado en 2004, fue todo un éxito aclamado por la crítica. Desde que se lanzó, hasta el día de hoy, ha sacado un total de seis expansiones para ir renovándose y mantener a los fans, sin que estos pierdan el interés.

### 2.3.2.8. MOBA

MOBA (Multiplayer Online Battle Arena), considerado un subgénero dentro de los juegos de estrategia, y más concretamente en los de estrategia de acción en tiempo real (ARTS), hablaremos de él como de un género más debido, sobre todo, al gran auge que ha tenido este tipo de juegos tanto en número de jugadores como en su importancia dentro de los *e-sports*.

La diferencia en cuanto a un juego de estrategia tradicional es que solamente manejas un personaje, el cual suele tener habilidades especiales y que pelea, generalmente contra otro equipo en igualdad numérica, al lado de otros

compañeros. En la mayoría, el objetivo del juego es destruir la base enemiga, aunque hay alguno en que simplemente es acabar con el equipo enemigo. También suele haber unidades más pequeñas manejadas por el juego que ayudan a los jugadores, comúnmente llamados “súbditos”, aunque no todos los tienen.

En cuanto al origen de este género, nos remontamos a la aparición de un mapa personalizado para StarCraft, llamado Aeon of Strife, en el cual cada jugador manejaba a una unidad más poderosa que el resto y la máquina manejaba al resto.

Después de esto, vino DotA, un mapa basado en lo mencionado anteriormente pero para otro juego, Warcraft III: Reign of Chaos, este es considerado el primer MOBA del que se empezaron a realizar torneos de los seguidores. A raíz del éxito de este novedoso estilo de juego se crearon otros, entre los que destaca League Of Legends y Dota 2.


*Figura 2.13. Partida retransmitida en directo de League Of Legends Fuente: [25]*

League Of Legends [26] se lanzó en 2009 y su dinámica principal es destruir la base enemiga. Para llegar a ella se destruyen tanto los súbditos, como torres o, los enemigos manejados por otros jugadores que nos vayamos encontrando. Dentro del juego hay distintos roles, y aunque no vamos a profundizar en ellos, nos parece relevante mencionar que para que un equipo esté balanceado debe haber personajes muy duros y lentos y otros muy ligeros y rápidos. Cada personaje tiene sus habilidades únicas que lo diferencian de los demás, y le ayudan a combatir a sus adversarios y/o ayudar a sus aliados.

Por el componente inherentemente competitivo que tiene este tipo de juegos, a menudo, 5 vs 5, y gana uno, como si de un partido de fútbol se tratase, ha llegado a ser el mayor referente dentro de los *e-sports*, elevando este fenómeno a unos niveles insospechados hace unos años.

### 2.3.3. Videojuegos multijugador y e-sports

Los deportes electrónicos o *e-sports* son competiciones de videojuegos multijugador en los que se enfrentan jugadores profesionales. Los géneros más comunes, y también, los más populares. Estos son: estrategia en tiempo real (StarCraft II), disparos en primera persona (Counter Strike: GS) y MOBA (League Of Legends y Dota 2).

Los *e-sports* [27] existen desde hace bastante tiempo, pues empezaron prácticamente con los juegos multijugador. El componente del reto entre los jugadores es un elemento que, tarde o temprano desembocaría en torneos. Al principio se trataba de competiciones locales, pero con la expansión de internet y las mejoras tanto en los ordenadores como en la calidad de los juegos, estos torneos fueron cada vez más grandes, llegando así a más gente. A partir del año 2012 aproximadamente, hubo un juego que empezó a sobrepasar todos los récords en cuanto a seguimiento y visualizaciones, el mencionado League Of Legends. Algunos informes reflejan que en 2011 contaba con 11.5 millones de usuarios, en 2014 ya eran 67, y en 2016 llegó a los 100, siendo la última cifra conocida, aunque viendo la proyección no extrañaría que los hubiera superado ampliamente. El mayor evento de este videojuego es el mundial (League Of Legends World Championship), en el cual, se reúnen los mejores equipos de las distintas regiones participantes (Corea, China, Europa, Norte América, etc).


*Figura 2.14. Estadio de las finales de los Worlds de League of Legends Fuente: [28]*

Este evento, en 2017, [29] pudo congregar a 80 millones de espectadores únicos en la semifinal (en la que se enfrentaban los dos favoritos) y a 56 millones en la final.

Podemos decir así que este juego, junto con otros no tan ampliamente seguidos, pero sí muy populares, han creado una industria que ha venido para quedarse, los *e-sports*. Tanto es así que, visto el éxito, muchas productoras de videojuegos se están esforzando por intentar enfocar sus juegos en un tono más competitivo, por si tuvieran la suerte de que algún día esos juegos también llegaran a ser un deporte electrónico.

Puede parecer que los *e-sports* se limitan a juegos de ordenador, pero no es así. Aunque la gran mayoría son para PC, últimamente salen nuevos *e-sports* muy seguidos en otras plataformas, como son los dispositivos móviles. Es el caso de Clash Royale.

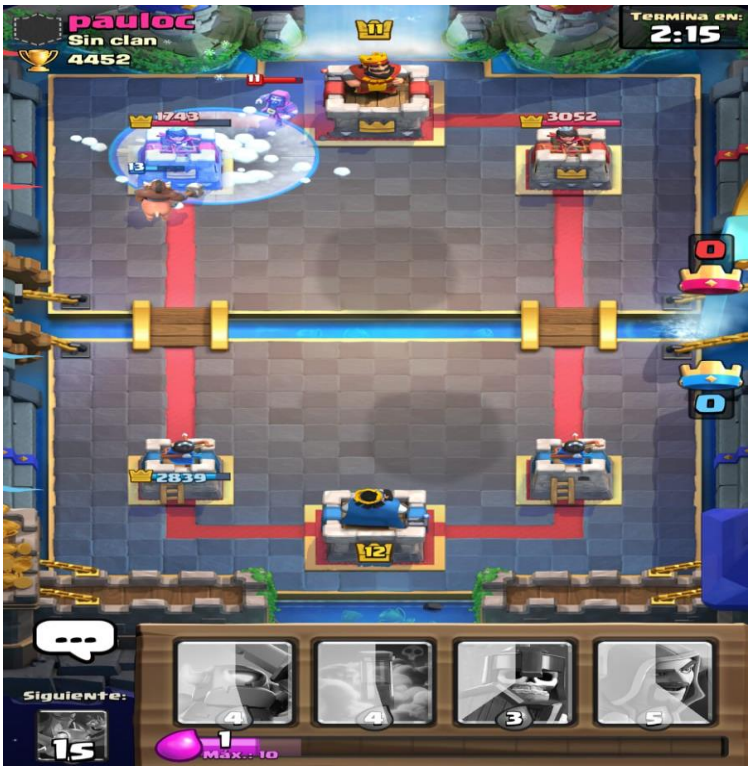


*Figura 2.15. Partida de Clash Royale*

Creado por Supercell, es un juego que mezcla varios géneros, entre ellos cartas coleccionables, defender la torre y estrategia en tiempo real. Sus plataformas son móviles y tabletas, siendo un juego que se juega de manera táctil, indicando con el dedo dónde quieres desplegar las unidades. El objetivo es tirar más torres al final de la partida o acabar con la torre central de tu rival antes que él. Mecánica sencilla, pero con muchos detalles y unidades (cartas) muy variadas. Esta empresa se ha animado recientemente (diciembre 2018) a lanzar su propio MOBA, Brawl Stars.

No es un MOBA tan complejo como League Of Legends, los juegos de móviles tienen sus particularidades y problemática concreta, al igual que tienen sus puntos a favor.

Debido a las limitaciones que tiene la pantalla, hay que aprovechar muy bien el espacio disponible para que el jugador pueda manejarse libremente, sin que le quiten mucha visibilidad sus propios dedos. Otra cosa para tener en cuenta son los gráficos y si el juego es 2D o 3D, ya que los móviles no tienen la misma capacidad de procesamiento que un PC.

En cuanto a los pros, tenemos la gran accesibilidad al juego, ya que cualquier usuario lo puede encontrar en la tienda de su smartphone y, si el juego no requiere mucho, podrá ser jugado en prácticamente cualquier dispositivo. Otro punto importante es la inmediatez. Simplemente pulsar el icono del juego en tu smartphone para empezar a jugar, en cambio, los juegos de PC no son tan inmediatamente accesibles.

Podemos concluir que la sencillez y accesibilidad es la clave para conquistar los *e-sports* en los *smartphones*.

## 2.4. Motores de videojuegos

Para ayudar a los desarrolladores a realizar los videojuegos, se dispone de unas herramientas específicas llamadas motores de videojuegos (*game engine*). Estos entornos de desarrollo proveen herramientas que se centran en los aspectos más característicos de los videojuegos, aspectos que otro tipo de software no necesita.

Entre estas funcionalidades podemos encontrar:

• Renderizado de gráficos 2D y 3D, una de las más importantes ya que sin ella no podríamos ver correctamente el videojuego.

• Librerías de simulación de fuerzas físicas, que imite, en caso de que se quiera, las leyes de la física. También sirven, por ejemplo, para detectar colisiones entre elementos.

• Animación, tanto en 2D como en 3D, que permite que cualquier elemento del entorno del videojuego sea animado, ayudado por los renderizadores.

• Scripting, permite que los elementos del videojuego, creados con el editor, como pueden ser figuras o personajes, tengan un comportamiento específico designado por un script.

• Sonidos, permite incluir sonidos para que el videojuego los reproduzca en momentos determinados.

A parte de las mencionadas, según el motor gráfico que escojamos, se nos puede proporcionar otros aspectos que también pueden ser útiles, como inteligencia artificial o el uso de redes. A continuación, vamos a ver un poco de los más importantes.

### 2.4.1. Unity

Unity [30] [31] es un motor de videojuego multiplataforma desarrollado por Unity Technologies. Es una de las opciones más recomendadas para principiantes, debido a que la curva de aprendizaje es moderada, además de que hay una comunidad amplia que puede ayudar a resolver las dudas. Se recomienda para hacer juegos relativamente sencillos para múltiples plataformas.


*Figura 2.16. Logo de Unity*

### 2.4.2. Unreal Engine

Unreal Engine es un motor de videojuego desarrollado por Epic Games. Es una herramienta usada hoy día por un gran número de desarrolladores de juegos. Actualmente en su versión 4, está considerado el motor más potente del mercado, permitiendo crear juegos con un nivel de realismo muy elevado. La curva de aprendizaje es más complicada y la comunidad no es tan grande como la de Unity.


*Figura 2.17. Logo de Unreal Engine*

### 2.4.3. Elección del motor

Hay muchos más motores gráficos, pero los dos más importantes y con más comunidad, son los indicados. Así que vamos a hacer una comparativa para ver cuál es el que más se ajusta al proyecto [32].

Como hemos mencionado anteriormente, unas de las mejores bazas de Unity es su curva de aprendizaje y la comunidad que posee, la cual hace aún más asequible el aprendizaje. Unreal Engine en cambio tiene una curva algo más difícil, y la comunidad no es tan amplia como la de Unity.

En cuanto al tema de la potencia, Unreal es bastante más potente que Unity, permite hacer juegos más complejos y conseguir unos resultados muy realistas. Unity, en cambio, está orientado a juegos más sencillos. Esto es debido a que la potencia de Unreal, depende del rendimiento del dispositivo que lo va a ejecutar. Como en nuestro caso va a ser un juego sencillo, en 2D, y para móvil (rendimiento del dispositivo limitado), en este sentido, Unity es el más acertado.

Orientándonos a una realización para dispositivos móviles, la gran integración multiplataforma de Unity nos hará sencillo el desarrollarlo para, por ejemplo, Android.

Otro punto a favor de Unity es que posee una gran tienda de componentes (Asset Store), en la cual podemos encontrar multitud de elementos que nos pueden ayudar a realizar nuestro videojuego.

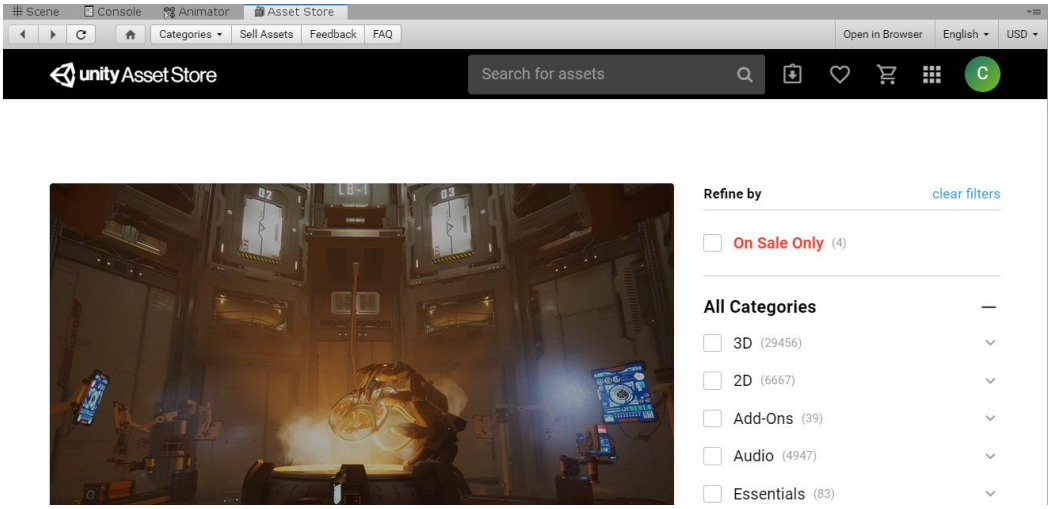


*Figura 2.18. Asset Store de Unity*

Desde imágenes, animaciones y sonidos hasta desarrollos de otros usuarios con algún propósito específico que nos pueden ayudar al integrarlo en nuestro proyecto.

Otro punto a favor de Unity es que dispone de una API y varios sistemas de apoyo para realizar juegos en red, como por ejemplo un servidor gratuito para alojar las partidas de nuestro juego. Este sistema se llama UNET [33] y usa el protocolo UDP para muchas de sus funcionalidades.

Por todo esto, elegiremos Unity como motor para la realización de nuestro videojuego.

### 2.4.4. Tecnologías complementarias

Además de todas las facilidades que nos brinda Unity, de las que veremos el resultado de su uso en la parte de implementación, será necesario utilizar otro software para otro tipo de tareas necesarias para el desarrollo completo del videojuego, tales como la creación y edición de imágenes o la realización de diagramas para las fases previas a la implementación.

#### 2.4.4.1. GIMP

GIMP (GNU Image Manipulation Program) es un editor de imágenes multiplataforma. Es un programa libre y gratuito, forma parte del proyecto GNU

y está disponible bajo la licencia pública de GNU. Provee de sofisticadas herramientas para la edición de imágenes que se pueden asemejar a sus competidores de pago.

*Figura 2.19. Logo de GIMP*

Principalmente se usará para el retoque de ciertas imágenes, por ejemplo, para los componentes de los menús, del HUD, etc.

#### 2.4.4.2. Piskel

Piskel es un editor online para la realización de *sprites* (imágenes orientadas a videojuegos) y animaciones en estilo “PixelArt”, esto es, dibujado y coloreado píxel a píxel.

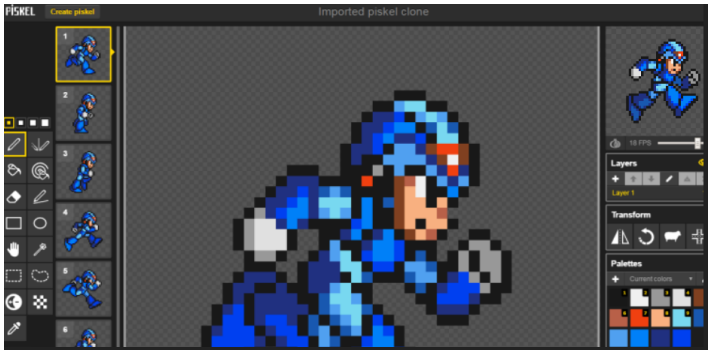


*Figura 2.20. Piskel*

Lo usaremos para la creación de algunos objetos o incluso proyectiles desde cero, para incluir en nuestro juego.

#### 2.4.4.3. Visual Studio

Visual Studio es un entorno de desarrollo integrado desarrollado por Microsoft. Es uno de los entornos que tiene integración con Unity y de los cuales se recomienda su uso para programar los scripts de Unity.


*Figura 2.21. Logo de Visual Studio*

Será el único entorno de desarrollo que usaremos para programar. Su integración con Unity hace muy cómodo el desarrollo.

#### 2.4.4.4. Astah

Astah es una herramienta de diseño de software que soporta UML, ERD y DFD entre otros.


*Figura 2.22. Logo de Astah*

Lo usaremos tanto para los diagramas de los casos de uso, las descripciones de éstos, y todos los diagramas realizados en la fase de diseño.

#### 2.4.4.5. MS Project

MS Project es una herramienta software de gestión de proyectos desarrollado por Microsoft. Lo usaremos para definir la planificación del proyecto.


*Figura 2.23. Logo de MS Project*

# 3. Documento de Diseño del Juego

El Documento de Diseño del Juego [34] o GDD (Game Design Document) es un documento cuyo propósito pretende ser el de hacer una descripción detallada del videojuego a desarrollar. La realización de este documento pasa por las manos de los diseñadores, analistas y programadores. Básicamente, es un documento de diseño, como si de una aplicación normal se tratara, pero adaptado a las singularidades que solo los videojuegos poseen. Estas singularidades son aspectos solo relevantes para los videojuegos que, en otro tipo de software, en ocasiones, ni siquiera existen.

El GDD consta de:

## 3.1. Título

Battle In The Sand (BITS)

## 3.2. Plataforma

Smartphone con Android superior a 4.0

## 3.3. Sinopsis de jugabilidad y contenido

El videojuego será multijugador online, de lucha en arena al estilo MOBA. El jugador dispondrá de varias habilidades que podrá lanzar a sus enemigos (otros jugadores) para conseguir alzarse como último en pie. Durante la partida, aparecerán en el suelo distintos objetos que, al recogerlos, alterarán ciertos parámetros de los jugadores para ayudarles (o perjudicarles) en su objetivo. Fuera de la partida, podrás configurar las habilidades que quieres que tenga tu personaje. Por ejemplo, es posible personalizar un kit de habilidades en los que

el personaje sea muy rápido, para esquivar mejor los ataques rivales, o también es posible centrarse en tener mucho daño pero menos velocidad.

## 3.4. Categoría

El juego estaría dentro de la categoría MOBA. Se asemejaría en la jugabilidad a juegos de móvil como Brawl Stars, pero, en cuanto al tema de las habilidades, también tiene características de otros juegos como puede ser League of Legends.

## 3.5. Licencia

Es de invención propia, un juego original.

## 3.6. Mecánica

### 3.6.1. Cámara

El juego será 2D y la cámara se encontrará desde un plano cenital lateral, similar a las retransmisiones de un partido de futbol, comúnmente conocida como *Top-Down*, en la que la mayoría de los elementos se ven desde arriba, pero los personajes pueden verse desde un lateral.

### 3.6.2. Controles

Se dispondrá de un *joystick* a la parte izquierda de la pantalla para el movimiento del personaje y 3 pequeños *joysticks* para las habilidades a la derecha de la pantalla.

### 3.6.3. Puntuación

No hay sistema de puntuación, el jugador que consiga derrotar al resto de sus adversarios, gana la partida. A no ser que se acabe el tiempo límite, en cuyo caso ninguno ganará.

### 3.6.4. Guardar/Cargar

Al ser un juego online, todos los progresos se irán guardando en el perfil online del jugador, alojado en la nube. Las preferencias de los usuarios con respecto a las habilidades se guardarán en el dispositivo local.

### 3.6.5. HUD

HUD (Head-Up Display) es la información que se muestra de manera continua mientras el usuario juega. Se dispondrá de información en el HUD sobre: el tiempo desde que empezó la partida, los atributos del jugador, las habilidades, los *PlayerModifier* que afectarán al jugador y durante cuánto tiempo.

## 3.7. Estados del juego

### 3.7.1. Pantalla de Identificación

Pantalla sencilla en la que se permitirá al usuario introducir su nombre de usuario para realizar la identificación.

### 3.7.2. Menú principal

Menú principal del videojuego en los que se podrá elegir entre Personalizar y Jugar (entrar al *lobby*).

### 3.7.3. Personalizar

En esta escena se podrán personalizar las habilidades que se quieran para el personaje. Entrando posteriormente en otra escena para personalizar exactamente el kit de habilidades deseado.

### 3.7.4. Lobby

*Lobby*, en español, vestíbulo, es como se conoce a las escenas de este tipo. En esta escena habrá una serie de menús y botones para permitir a los usuarios crear y encontrar partidas a las que unirse para poder jugar posteriormente.

### 3.7.5. Jugar

Una vez iniciada la partida configurada anteriormente en el Lobby, en esta escena directamente los jugadores podrán jugar libremente hasta que alguno se proclame vencedor.

## 3.8. Tecnología

Motor gráfico Unity, lenguaje de programación C#.

## 3.9. Público

El juego va orientado a jóvenes de edades comprendidas entre 13-25 años. Que dispongan de *smartphones* y tarifa de datos o wifi para poder jugar con sus amigos.

# 4. Desarrollo del proyecto

En este capítulo se habla sobre el desarrollo del proyecto y todos sus aspectos. Hablaremos del proceso de software elegido para llevar a cabo el proyecto, la planificación, así como los riesgos a los que nos podemos enfrentar.

• Documento de Gestión de Riesgos, en el que se detallan los riesgos que hemos podido encontrar en el inicio del proyecto, así como intentar prever los que puedan surgir a lo largo del desarrollo de este.

• Planificación del proyecto, en el que se expone el proceso de desarrollo elegido, características y motivos de la decisión. Se realiza la subdivisión de las tareas del proyecto y la planificación en el tiempo de ellas. Por último, el seguimiento del trabajo y la planificación final.

## 4.1. Gestión de Riesgos

A la hora de planificar un proyecto [35] [36], hay que tomar ciertas decisiones. En un escenario ideal, contaríamos con toda la información necesaria para poder tomarlas y sabríamos a ciencia cierta cuál sería el resultado de hacerlo. Desgraciadamente el mundo real no es así, y la mayoría de las decisiones que se pueden tomar, están basadas en información incompleta. La falta de información y la incertidumbre nos lleva a los riesgos, que son una parte inherente e intrínseca de la gestión de proyectos.

Cada vez que queramos añadir algo que, aunque sea una gran idea, sumará peso al proyecto, habrá riesgos asociados a ello. Riesgos que, de no ser tenidos en cuenta y producirse, pueden hacer peligrar el proyecto.

La Gestión de Riesgos de Proyecto es definida por el PMBOK como:

> *'… el proceso sistemático de identificar, analizar y reaccionar a los riesgos del proyecto… [durante todo el ciclo de vida del proyecto].*

También define un riesgo como:

> *'… factores que pueden causar un fracaso para cumplir los objetivos del proyecto…'.*

Una parte importante de la gestión de riesgos considerar qué probabilidad y qué impacto tendrá cada uno, para poder diseñar y priorizar los planes de mitigación y contingencia en consecuencia.

| | **Probabilidad de ocurrencia del escenario** | Muy baja | Baja | Media | Alta | Muy alta |
|---|---|---|---|---|---|---|
| **Impacto en el proyecto** | Muy bajo | 0 | 1 | 2 | 3 | 4 |
| | Bajo | 1 | 2 | 3 | 4 | 5 |
| | Medio | 2 | 3 | 4 | 5 | 6 |
| | Alto | 3 | 4 | 5 | 6 | 7 |
| | Muy alto | 4 | 5 | 6 | 7 | 8 |

*Tabla 4.1. Tabla de colores según impacto y probabilidad de riesgo*

Describimos a continuación los atributos que tendrá cada uno de los riesgos que encontremos.

• Categoría:

La categoría en la que se ve enmarcado el riesgo, puede ser de proyecto, de proceso o de producto.

• Riesgos de Proyecto

Restricciones de recursos, problemas de coordinación interna del equipo o del grupo, financiación no adecuada.

• Riesgos de Proceso

Proceso software pobremente documentado, no prevención de defectos, proceso de diseño insuficiente, gestión escasa de requisitos, planificación ineficaz.

• Riesgos de Producto

Falta de experiencia en el dominio de la materia, diseño demasiado complejo, interfaces definidas deficientemente, requisitos vagos o incompletos.

• Probabilidad:

Probabilidad de que ese riesgo ocurra, teniendo en cuenta el entorno y los distintos factores a los que se pueda exponer el proyecto. Según la probabilidad, podemos distinguirlo en:

**Muy alta**: Más de 80% de probabilidades de que ocurra.

**Alta**: Entre 60% y 80% de probabilidades de que ocurra.

**Media**: Entre 40% y 60% de probabilidades de que ocurra.

**Baja**: Entre 20% y 40% de probabilidades de que ocurra.

**Muy baja**: Menos de 20% de probabilidades de que ocurra.

• Impacto:

El impacto que tendría en el proyecto el riesgo en caso de producirse. Puede ir desde un impacto prácticamente despreciable a uno crítico o incluso catastrófico. El impacto que suele tener un riesgo se ve afectado a nivel de costes y a nivel de planificación. Según el nivel de impacto podemos distinguirlo en:

**Muy alto**: Más de 20% de sobrecostes sobre la planificación y estimación inicial.

**Alta**: Entre un 15% y un 20% de sobrecostes sobre la planificación y estimación inicial.

**Media**: Entre 10% y 15% de sobrecostes sobre la planificación y estimación inicial.

**Baja**: Entre 5% y 10% de sobrecostes sobre la planificación y estimación inicial.

**Muy baja**: Menos de 5% de sobrecostes sobre la planificación y estimación inicial.

• Descripción:

Breve, pero precisa descripción del riesgo que se ha descubierto.

• Fases afectadas:

Las fases del proceso unificado que pueden ser afectadas por el riesgo, desde iniciación hasta transición.

• Estrategia:

Cómo afrontar el riesgo: podemos intentar evitarlo, podemos intentar protegernos de ello, podemos investigarlo o, incluso aceptarlo.

• Plan de mitigación:

Las estrategias del plan de mitigación [37] tratan de reducir la probabilidad de ocurrencia del riesgo, o bien, reducir el impacto que pueda causar. Están referidas al conjunto de acciones que se toman por adelantado o acciones proactivas. Es decir, las medidas para intentar que el riesgo finalmente no ocurra.

• Plan de contingencia:

El PMBOK define estrategias del plan de contingencia a aquellas respuestas que se utilizan solamente si ocurre el riesgo. Planes alternativos para que, en caso de que el riesgo finalmente ocurra, afecte en lo mínimo posible la planificación inicial del proyecto.

Estos dos planes no son excluyentes. De hecho, es necesario planificar tanto el cómo intentar mitigar la posible ocurrencia de un riesgo como, en el caso de que éste finalmente ocurra, cómo poder corregirlo de la mejor forma posible y que afecte en la menor medida la planificación inicial del proyecto.

### 4.1.1. Riesgos encontrados

La identificación de los riesgos es probablemente la parte más difícil e importante del proceso de gestión de riesgos. Si no eres capaz de identificar un riesgo, va a ser excluido, y por tanto no sometido a un análisis más detallado, por lo que probablemente, de producirse, no se estaría preparado para responder a él. Una de las mayores fuentes de información para la detección de riesgos es la experiencia.

Al ser este el primer proyecto realizado en el motor gráfico Unity y con varias tecnologías cuyo funcionamiento se irá, o bien descubriendo de cero o bien profundizando más en una materia en la que se está solo iniciado, la experiencia es prácticamente nula y no podemos saber a qué riesgos específicos se enfrenta el proyecto. Lo que sí podemos saber es que esta falta de experiencia es un riesgo en sí mismo. Así que consideraremos como riesgos la falta de experiencia y los problemas que pueda acarrear ésta para el desarrollo del proyecto.

| **RIE-001** | **Problemas de implementación de Unity** | |
|---|---|---|
| **Categoría** | Producto | 5 |
| **Probabilidad** | Media | |
| **Impacto** | Alto | |
| **Descripción** | Debido a la inexperiencia con el motor Unity, podrían darse problemas de implementación y no cumplirse los objetivos relativos a experiencia de juego que se deseaban en un principio. | |
| **Fases afectadas** | Construcción | |
| **Gestión de riesgo** | | |
| **Estrategia** | Formación e investigación en las herramientas de Unity y en las características concretas que se desean implementar. | |
| **Plan de mitigación** | Realización de tutoriales específicos de esas características para intentar dominar las técnicas de implementación necesarias. | |
| **Plan de contingencia** | Búsqueda de otra forma de implementar esas características, con otras técnicas, para que se parezcan lo más posible a lo ideal. | |

*Tabla 4.2. RIE-001. Problemas de implementación de Unity*

| RIE-002 | Cambios en los requisitos | |
|---|---|---|
| **Categoría** | Proceso | 4 |
| **Probabilidad** | Media | |
| **Impacto** | Medio/Alto | |
| **Descripción** | Durante todo el proceso de desarrollo pueden darse cambios en los requisitos iniciales. A más tarde se decidan cambiar los requisitos, más alto será el impacto en el proyecto. | |
| **Fases afectadas** | -Elaboración<br>-Construcción | |
| **Gestión de riesgo** | | |
| **Estrategia** | Protegernos del riesgo | |
| **Plan de mitigación** | Enfoque en las fases más iniciales en delimitar bien los requisitos y que, de haber cambios, sean menores y sin mucha importancia. | |
| **Plan de contingencia** | Realización de las modificaciones tanto en la planificación como en la implementación para ajustar estos a los nuevos cambios. | |

*Tabla 4.3. RIE-002. Cambios en los requisitos*

| RIE-003 | Errores o contenido ambiguo del diseño | |
|---|---|---|
| **Categoría** | Proceso | 3 |
| **Probabilidad** | Baja | |
| **Impacto** | Medio | |
| **Descripción** | Que los documentos de diseño finales no sean lo suficientemente concretos y descriptivos, puede llevar a errores en la implementación final. | |
| **Fases afectadas** | -Elaboración<br>-Construcción | |
| **Gestión de riesgo** | | |
| **Estrategia** | Protegernos del riesgo | |
| **Plan de mitigación** | Enfocarse en tener algo muy concreto en los documentos de diseño y, en las propias etapas de Construcción, ir concretando más y más a medida que avanza el proyecto. | |
| **Plan de contingencia** | Replantear ciertas bases del documento de diseño en la fase de Elaboración para darle un nuevo enfoque, intentando evitar lo que se ha diagnosticado que ha dado problemas en primera instancia. | |

*Tabla 4.4. RIE-003. Errores o contenido ambiguo del diseño*

| RIE-004 | Indisponibilidad del desarrollador | |
|---|---|---|
| **Categoría** | Proyecto | 2 |
| **Probabilidad** | Baja | |
| **Impacto** | Bajo | |
| **Descripción** | Cabe la posibilidad de que el desarrollador caiga enfermo y éste no pueda realizar su actividad habitual durante varios días, lo que puede suponer algún retraso en la planificación. | |
| **Fases afectadas** | -Elaboración<br>-Construcción<br>-Transición | |
| **Gestión de riesgo** | | |
| **Estrategia** | Aceptar el riesgo | |
| **Plan de mitigación** | | |
| **Plan de contingencia** | Dependiendo del grado de indisposición del desarrollador, podrá no realizar ninguna actividad, o quizás puede realizar las que menos esfuerzo le requieran, intentando así que la planificación se vea lo menos afectada posible. En caso de que no pueda realizar ninguna tarea relativa al proyecto, habría que reajustar la planificación. | |

*Tabla 4.5. RIE-004. Indisponibilidad del desarrollador*

| RIE-005 | Avería en el ordenador | |
|---|---|---|
| **Categoría** | Proyecto | 3 |
| **Probabilidad** | Muy baja | |
| **Impacto** | Alto | |
| **Descripción** | El ordenador que se está utilizando para el desarrollo del software puede sufrir algún tipo de avería. | |
| **Fases afectadas** | -Elaboración<br>-Construcción | |
| **Gestión de riesgo** | | |
| **Estrategia** | Protegernos del riesgo | |
| **Plan de mitigación** | Tener otro ordenador de características similares disponible para poder continuar realizando el trabajo sin que, o afectando lo mínimo posible, la planificación inicial. | |
| **Plan de contingencia** | Empezar a usar el ordenador mencionado anteriormente y continuar realizando el trabajo. | |

*Tabla 4.6. RIE-005. Avería en el ordenador*

| RIE-006 | Problemas con el servidor | |
|---|---|---|
| **Categoría** | Producto | 4 |
| **Probabilidad** | Alta | |
| **Impacto** | Bajo | |
| **Descripción** | Puede que haya problemas con el servidor disponible para hostear la partida durante ésta, arruinando así la experiencia de juego. | |
| **Fases afectadas** | Transición | |
| **Gestión de riesgo** | | |
| **Estrategia** | Protegernos del riesgo | |
| **Plan de mitigación** | Se intentará tener disponible un servidor que garantice una disponibilidad y calidad del servicio aceptable para poder ofrecer una buena experiencia de juego (poca latencia y alta disponibilidad). | |
| **Plan de contingencia** | Si el servidor actual da demasiados problemas y no cumple con lo establecido en el plan de mitigación, se procedería a contratar otro servidor. | |

*Tabla 4.7. RIE-006. Problemas con el servidor*

| RIE-007 | Pérdida de información | |
|---|---|---|
| **Categoría** | Proceso | 3 |
| **Probabilidad** | Muy Baja | |
| **Impacto** | Alto | |
| **Descripción** | Cabe la posibilidad de que, por accidente, descuido o fallo informático, se produzca una pérdida de información. Estas pérdidas pueden tener distintas magnitudes. Puede perderse prácticamente todo el proyecto o puede perderse simplemente el trabajo de las últimas dos horas. | |
| **Fases afectadas** | Elaboración y Construcción | |
| **Gestión de riesgo** | | |
| **Estrategia** | Protegernos del riesgo | |
| **Plan de mitigación** | Se irán realizando copias de seguridad en la nube de forma periódica para evitar la pérdida de información. Esta periodicidad puede tener mayor frecuencia si se observa que el trabajo realizado al final del día ha sido lo suficientemente importante como para realizar copias de seguridad extraordinarias. | |
| **Plan de contingencia** | En caso de haber pérdida de información se procedería a recuperar la última copia de seguridad para minimizar el impacto. | |

*Tabla 4.8. RIE-007. Pérdida de información*

| RIE-008 | Problemas derivados de incluir red en el juego | |
|---|---|---|
| **Categoría** | Proceso | 5 |
| **Probabilidad** | Alta | |
| **Impacto** | Medio | |
| **Descripción** | Puede ocurrir que se realice una implementación primeramente enfocada en que funcione en modo 1 jugador, y que, a la hora de incluir más jugadores no funcione correctamente en un principio. | |
| **Fases afectadas** | Implementación | |
| **Gestión de riesgo** | | |
| **Estrategia** | Aceptar el riesgo y tratar de que afecte lo menos posible | |
| **Plan de mitigación** | Se intentará realizar la implementación teniendo en cuenta que debe funcionar para la red, evitando formas de implementación que excluya el uso de red, para así que el "paso a red" sea un sencillo paso más, no un gran problema que requiera mucho cambio en el código. | |
| **Plan de contingencia** | Se cambiará lo pertinente en el código, esperando que no sea mucho lo que haya que cambiar y que no afecte mucho en la planificación. En caso de que sí afecte significativamente, probablemente hubiera que modificar la planificación. | |

*Tabla 4.9. RIE-008. Problemas derivados de incluir red en el juego*

## 4.2. Roles y funciones de los recursos disponibles

A continuación, describimos los roles que desempeña cada uno de los participantes en el proyecto y sus responsabilidades.

| Participante: Margarita Gonzalo Tasis | |
|---|---|
| **Rol** | **Responsabilidades** |
| **Cliente** | Es el responsable, junto con el analista, de fijar los objetivos y requisitos del sistema. También ayuda en la toma de decisiones al equipo de desarrollo. |
| **Asesor** | Se encarga de orientar y asesorar el proceso de desarrollo procurando que sigan el camino correcto en todas las fases del proyecto, basándose en su experiencia. |

*Tabla 4.10. Tabla de roles. Participante: Margarita Gonzalo Tasis*

| Participante: Carlos López Garcinuño | |
|---|---|
| **Rol** | **Responsabilidades** |
| **Jefe de Proyecto** | Entre las funciones del jefe de proyecto están planificar, coordinar y gestionar los recursos de los que dispone de tal forma que el proyecto sea eficiente en cuanto a sus costes y respetar los plazos. |
| **Analista** | La función principal del analista es recabar información junto con el cliente para poder realizar el listado de los requisitos y los casos de uso del sistema. |
| **Diseñador de Software** | Con la información recibida por parte del analista, el trabajo del diseñador es realizar el diseño de la arquitectura del sistema y del modelado del dominio, así como preparar el terreno para el trabajo del programador con los diagramas de interacción. |
| **Programador** | Programará la lógica interna de la aplicación y la interfaz de usuario. |
| **Diseñador gráfico** | Diseña, modifica y crea los distintos recursos que serán necesarios para el apartado visual del juego. |
| **Verificador** | Su función es la de realizar las pruebas con el objetivo de encontrar posibles fallos e informar de estos para que puedan ser resueltos rápidamente. |

*Tabla 4.11. Tabla de roles. Participante: Carlos López Garcinuño*

## 4.3. Planificación del proyecto

### 4.3.1. Proceso de desarrollo

A la hora de iniciar la planificación de un proyecto, es vital elegir correctamente las metodologías usadas para desarrollarlo. El Proceso Unificado (UP) es un proceso de desarrollo de software que está dirigido por los casos de uso y es iterativo e incremental.

Debido a que nuestro proyecto va a ser un videojuego prácticamente jugable desde las fases más tempranas de su desarrollo, hemos pensado que someterlo a un proceso iterativo e incremental es un acierto. Una de las bondades que propone el proceso unificado es que, podemos empezar con un prototipo bastante simple, y a partir de ahí ir refinando la parte correspondiente de cada fase de desarrollo, para que se ajuste correctamente a lo deseado, cambiando incluso ciertas decisiones que se tomaron en un principio pero que posteriormente se ha encontrado una solución mejor, sin que esto afecte negativamente a toda la planificación del proyecto, ya que estos deseos de cambios se detectarían a tiempo.

Las fases, debido al proceso unificado elegido, son las siguientes, aunque se hayan descompuesto en tareas más pequeñas dentro de cada fase.

-**Fase de Iniciación**:

En esta fase definimos el proyecto, establecemos el alcance y los objetivos de este. Los documentos de salida de esta fase son el documento de la planificación del proyecto y el de gestión de riesgos.

-**Fase de Elaboración**:

En esta fase se realiza la identificación de los requisitos, casos de uso de nuestra aplicación y diseño de la arquitectura del sistema. Podemos diferenciar dos fases dentro del proceso de elaboración. La primera, algo más abstracta, consiste en la toma de requisitos y casos de uso, lo que se conoce más como la fase de análisis. Y la segunda parte, más concreta, consiste en el modelado y diseño del sistema.

**-Fase de Construcción**:

En esta fase se lleva a cabo la implementación completa del software de la aplicación, así como una serie de pruebas durante el desarrollo. Al finalizar esta fase deberían estar acabados la totalidad de las funcionalidades descritas en los casos de uso recabados en la fase de elaboración.

-**Fase de Transición**:

Finalmente, en esta fase, se llevan a cabo las pruebas finales y la corrección de los últimos detalles. Al final de la fase, tendremos varios documentos como el documento de pruebas, manuales de instalación y uso.

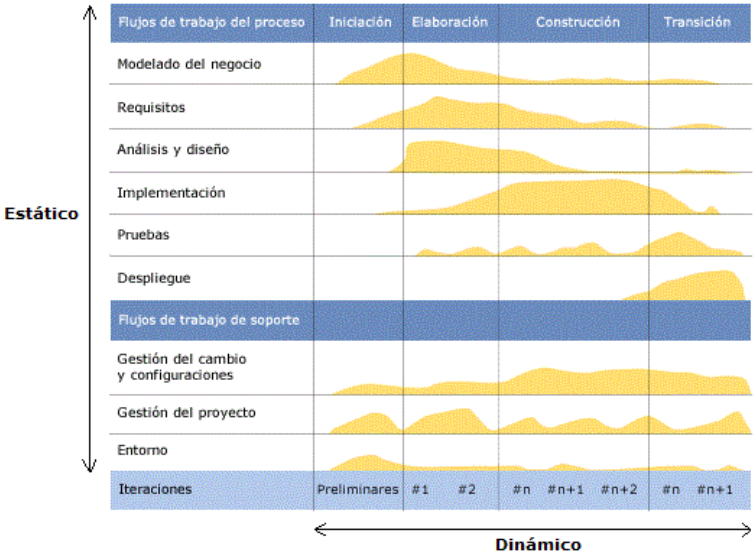


*Figura 4.1. Distribución de carga de trabajo por fases e iteraciones en UP Fuente: [38]*

En la figura anterior podemos ver cómo funciona el Proceso Unificado. En las primeras iteraciones, como es lógico, se presta más atención y la carga de trabajo está más enfocada en las tareas de toma de requisitos y análisis, pero sin dejar de lado el resto de las fases. Poco a poco, según avanzan las iteraciones, se va reduciendo el peso en las tareas de requisitos y análisis y se va centrando el enfoque en diseño e implementación. En fase de Construcción, la implementación ocupa la gran mayoría de la carga de trabajo, pero nunca se olvida las fases anteriores, que siguen refinando y mejorando el producto. Como para el final del proceso de desarrollo, como es normal, tanto las primeras fases como, incluso la implementación, tienen poco peso y todo se centra en las pruebas y el despliegue.

### 4.3.2. Fases del proyecto

Describiremos con un diagrama de Gantt, realizado con MS Project, las distintas fases planificadas para el desarrollo del proyecto. En el diagrama aparecen, dependiendo de la fase, las fases divididas en tareas más pequeñas o, como es el caso de construcción, directamente sus tres iteraciones. Los contenidos e hitos de las iteraciones de construcción lo explicaremos más adelante. Antes de explicar en detalle cada fase e iteración, vamos a ver una vista general del proyecto [39] .

| Fase | Iteración | Semanas | Fecha Inicio | Fecha Fin |
|---|---|---|---|---|
| Inicio | 1 | 1 | 22/02/19 | 26/02/19 |
| Elaboración | 2 | 5 | 27/02/19 | 02/04/19 |
| Construcción | 3 | 11 | 03/04/19 | 19/06/19 |
| Transición | 1 | 1 | jue 20/06/19 | 26/06/19 |

*Tabla 4.12. Vista general de las fases del proyecto*

Vamos a explicar las fases e iteraciones principales en las que se va a dividir el proyecto:

• **Fase 1**: Primera iteración de Inicio. En esta fase se desarrollará el documento de diseño del videojuego (GDD).

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Iniciación | 3 días | vie 22/02/19 | mar 26/02/19 |

*Tabla 4.13. Contenido primera iteración de Inicio*

• **Fase 2**: Primera iteración de Elaboración. En esta fase se comenzará con la toma de requisitos y análisis de estos. Así como de la identificación de los casos de uso.

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Toma de requisitos | 2 días | mié 27/02/19 | jue 28/02/19 |
| Análisis de los requisitos | 2 días | vie 01/03/19 | lun 04/03/19 |
| Realización de los casos de uso | 1 día | mar 05/03/19 | mar 05/03/19 |
| Descripción detallada de los casos de uso | 3 días | mié 06/03/19 | vie 08/03/19 |

*Tabla 4.14. Contenido primera iteración de Elaboración*

• **Fase 3**: Segunda iteración de Elaboración. En esta fase se realizará el diseño del sistema.

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Realización del modelo de dominio | 3 días | lun 11/03/19 | mié 13/03/19 |
| Realización del modelo entidad-relación | 3 días | jue 14/03/19 | lun 18/03/19 |
| Modelo de dominio (Diseño) | 4 días | mar 19/03/19 | vie 22/03/19 |
| Diagramas de secuencia | 7 días | lun 25/03/19 | mar 02/04/19 |

*Tabla 4.15. Contenido segunda iteración de Elaboración*

• **Fase 4**: Primera iteración Construcción. Se comenzará con la formación del equipo en la herramienta Unity. En esta fase se marca como hito, y, por lo tanto, se implementará el movimiento del personaje con el *joystick*, así como la gestión de objetos y los modificadores de las estadísticas del usuario.

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Formación en Unity | 5 días | mié 03/04/19 | mar 09/04/19 |
| Construcción (Etapa 1) | 17 días | mié 10/04/19 | jue 02/05/19 |

*Tabla 4.16. Contenido primera iteración de Construcción*

• **Fase 5**: Segunda iteración Construcción. En esta fase se marca como hito, y, por lo tanto, se implementará el HUD y el lanzamiento de las habilidades, así como la personalización de los sets de habilidades.

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Construcción (Etapa 2) | 17 días | vie 03/05/19 | lun 27/05/19 |

*Tabla 4.17. Contenido segunda iteración de Construcción*

• **Fase 6**: Tercera iteración de Construcción. En esta fase se marca como hito, y, por lo tanto, se adaptará todo lo anterior a la red, así como la realización de la gestión de final de partida y emparejamiento para conformarlas.

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Construcción (Etapa 3) | 17 días | mar 28/05/19 | mié 19/06/19 |

*Tabla 4.18. Contenido tercera iteración de Construcción*

• **Fase 7**: Primera iteración de Transición. Se realizarán las pruebas unitarias para comprobar que todo el sistema funciona correctamente, además de los manuales.

| Nombre de tarea | Duración | Comienzo | Fin |
|---|---|---|---|
| Transición | 5 días | jue 20/06/19 | mié 26/06/19 |

*Tabla 4.19. Contenido primera iteración de Transición*

El diagrama de Gantt global del proyecto es el siguiente.

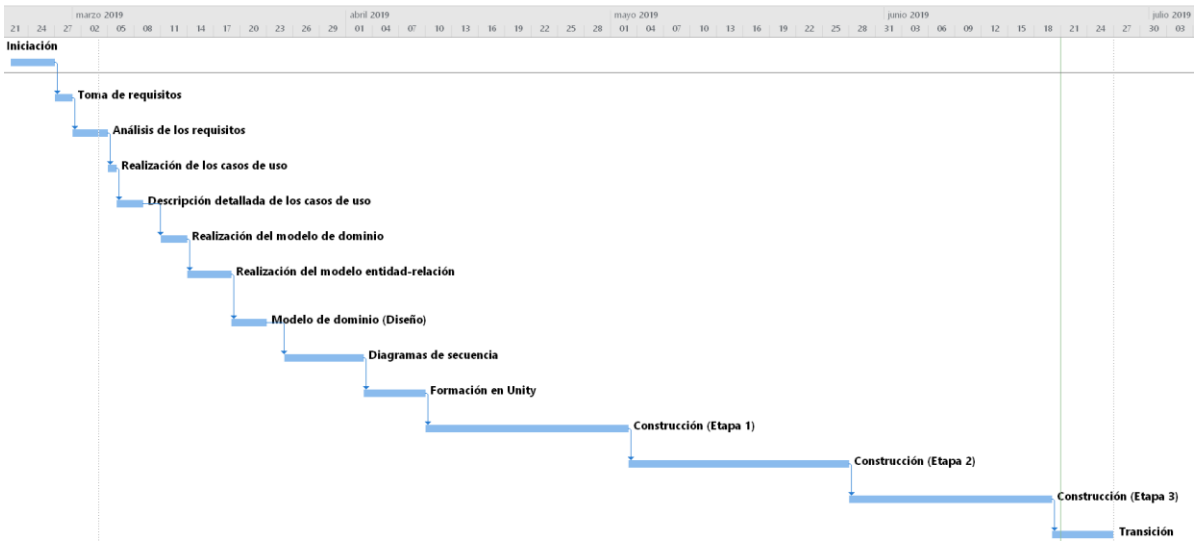


*Figura 4.2. Diagrama de Gantt del proyecto*

Como podemos ver en el diagrama de Gantt, comenzaríamos con la fase de iniciación que duraría unos 3 días. Después se llevará a cabo la primera fase de elaboración, que iría desde el Análisis de los requisitos hasta la Descripción detallada de los casos de uso. Le sigue la segunda fase de elaboración, que comprendería desde la Realización del modelo de dominio inicial hasta los Diagramas de secuencia. Una vez acabada la fase de Elaboración, comenzaremos la fase de Construcción, en la cual habrá tres fases, los hitos que denotan el final de cada fase los mencionaremos más adelante. En último lugar, se finalizará el proyecto con la fase de Transición.

Se calcula que el tiempo total del desarrollo del proyecto será de 17 semanas aproximadamente. Las horas de trabajo se contabilizarán de manera semanal, a un promedio de 18 horas de trabajo semanales. Resultaría un total de 306 horas.

### 4.3.3. Costes de producción previstos

Una vez realizada la planificación y su división en fases, así como la enumeración de los roles, vamos a proceder a estimar los costes totales del proyecto.

Independientemente de los roles que se vaya a tomar en el desarrollo del proyecto, en cuanto al coste de los recursos humanos vamos a calcular como si de un trabajador autónomo con titulación de ingeniero informático se tratara. Se estiman los honorarios por hora en 12€. Por lo tanto, en cuanto al coste de los recursos humanos tendremos un total de **3.672€.**

A parte de los costes humanos, también hay que tener en cuenta el coste material.

En nuestro caso, todo el software que hemos utilizado ha sido bien de uso libre, o bien bajo licencias provistas por la universidad, como es el caso de MS Project y Astah, por lo que no ha supuesto ningún coste añadido. En la siguiente tabla declararemos los gastos materiales asignados al proyecto. Como es obvio, no se espera que este proyecto asuma los costes íntegros de todos los recursos, por eso se ha calculado la parte proporcional a las 18 horas semanales que dedicaremos al proyecto.

| Recurso | Precio mensual | Coste total |
|---|---|---|
| Amortización del hardware | 30€ | 120€ |
| Conexión a internet | 10€ | 40€ |
| Alquiler | 80€ | 320€ |
| Suministro eléctrico | 12€ | 48€ |
| Material de oficina | 4€ | 16€ |

*Tabla 4.20. Recursos materiales*

El coste total de los recursos materiales y de alojamiento del proyecto asciende a **544€**. Si calculamos el coste total previsto del proyecto, obtenemos un coste estimado de **4.216€**.

### 4.3.4. Seguimiento del proyecto

A continuación, describimos cómo ha resultado la planificación final del proyecto, si nos hemos ajustado a los plazos, o, si ha aparecido algún riesgo y cómo lo hemos afrontado.

El proyecto comenzó según lo esperado, las primeras fases se realizaron de forma correcta, al menos sus primeras iteraciones. Sin embargo, en las primeras fases de construcción se materializó el riesgo (**RIE-001**), relacionado con los problemas de implementación con Unity. Era un riesgo que habíamos calculado con probabilidad media, ya que contábamos con que la curva de aprendizaje no sería complicada, y así lo fue. El problema fueron unos errores que el equipo no era capaz de solventar y mantuvo estancada la implementación algo más de una semana. Durante este tiempo, con el objetivo de mantener la planificación por fechas intacta, se incrementó el número de horas diarias hasta que el problema estuvo resuelto y se pudo volver a la normalidad. Una vez superada la curva de aprendizaje y asimilados los errores que ralentizaron el proyecto, fue todo bastante fluido. Se detectaron algunos errores, pero que no daban problemas más allá de un par de días, y que entraban dentro de lo planeado.

Durante la fase de construcción, se seguían refinando las fases de análisis y diseño, por suerte, ninguno de los cambios que se hicieron repercutió de manera negativa en el desarrollo del proyecto, ya que eran cuestiones más finas, que no se habían empezado a implementar aún, por lo que el riesgo de que afectaran negativamente (**RIE-002**) no llegó a cumplirse.

En cuanto al resto de los riesgos, ninguno llegó a materializarse. Afortunadamente no hubo pérdidas de información, aunque el plan de mitigación se continuó realizando hasta el final del proyecto. En cambio, en la tercera fase de construcción, ocurrió el riesgo relativo a los problemas de la implementación en red (**RIE-008**). La planificación inicial contaba con la alta probabilidad de ocurrencia de este riesgo. Gracias a que se contaba con ello, el equipo necesitó incrementar el número de horas diarias en menor medida que con el **RIE-001** ya que, al tener éste una probabilidad más alta, se calculó un tiempo de resolución más holgado.

La ampliación de horas en estas fases hizo aumentar la media de horas semanales dedicadas al proyecto hasta un total de 20. Por tanto, habría que recalcular los costes tanto humanos como materiales.

El nuevo cálculo de horas totales dedicadas al proyecto es de 340. Por tanto, el coste de recursos humanos asciende ahora a **4.080€**. También tenemos que aumentar la parte proporcional de los recursos materiales, lo que nos deja **604€**. Obtenemos entonces un coste final total de **4.684€**.

Debido a la inexperiencia en la planificación de proyectos y que se trata de un campo del que no se tenían apenas conocimientos previos, la planificación inicial no se ha cumplido correctamente y han surgido unos sobrecostes de **468€.**

# 5. Análisis

## 5.1. Propósito general

La aplicación deberá permitir al usuario jugar una partida de nuestro videojuego al pulsar el botón “Jugar”. Una vez pulsado, el sistema de emparejamiento buscará uno o varios jugadores disponibles y comenzará una partida entre todos ellos. Las mecánicas del juego serán descritas más en profundidad en otros apartados. Además de jugar, el sistema deberá permitir al usuario personalizar las habilidades del personaje con el que desea jugar, así como algunas opciones de sonido y video.

## 5.2. Alcance del sistema

El sistema realizará tanto el entorno y componentes del juego, como la navegación y lógica interna del mismo con ayuda del motor gráfico Unity.

La parte relativa a la conectividad multijugador se realizará con ayuda de la API que Unity tiene destinada para ello, UNET.

## 5.3. Requisitos del sistema

Un requisito [40] es una “condición o capacidad que necesita el usuario para resolver un problema o conseguir un objetivo determinado”. También se consideran requisitos las condiciones que debe poseer o cumplir un sistema para satisfacer un contrato, norma o especificación. Es decir, puede verse tanto como una declaración abstracta, comúnmente de alto nivel, de un servicio que el

sistema debe proporcionar, o como una definición matemática detallada de una función o restricción del sistema.

Vamos a serparar, en dos grupos, los requisitos funcionales y los no funcionales.

### 5.3.1. Requisitos funcionales

Los requisitos funcionales definen los servicios que el sistema debe proporcionar, cómo debe reaccionar a una entrada concreta y cómo se debe comportar según qué situaciones.

| **RF-001** | Crear partida |
|---|---|
| **Descripción** | El sistema deberá permitir al usuario crear una partida. |

*Tabla 5.1. RF-001. Crear partida*

| **RF-002** | Descubrir partida |
|---|---|
| **Descripción** | El sistema deberá permitir al usuario que descubra partidas que otros usuarios hayan creado. |

*Tabla 5.2. RF-002. Descubrir partida*

| **RF-003** | Unirse a una partida creada |
|---|---|
| **Descripción** | El sistema deberá permitir al usuario unirse a una partida creada por otro usuario. |

*Tabla 5.3. RF-003. Unirse a una partida creada*

| **RF-004** | Iniciar la partida |
|---|---|
| **Descripción** | El sistema deberá permitir al usuario pulsar el botón de “preparado” y, si el resto de los jugadores lo han pulsado también, dar comienzo a la partida. |

*Tabla 5.4. RF-004. Iniciar partida*

| **RF-005** | Finalizar partida |
|---|---|
| **Descripción** | El sistema deberá dar por acabada la partida cuando solo quede un jugador con vida superior a 0 o bien cuando acabe el tiempo límite. |

*Tabla 5.5. RF-005. Finalizar partida*

| RF-006 | Movimiento del personaje |
|---|---|
| Descripción | El sistema deberá permitir al personaje moverse (en todas direcciones) con ayuda de un *joystick* situado a la parte izquierda de la pantalla. |

*Tabla 5.6. RF-006. Movimiento del personaje*

| RF-007 | Lanzamiento de habilidades |
|---|---|
| Descripción | El sistema deberá permitir al personaje lanzar las habilidades que tenga (en todas direcciones) con ayuda de un *joystick* situado a la parte derecha de la pantalla. |

*Tabla 5.7. RF-007. Lanzamiento de habilidades*

| RF-008 | Impactar habilidades |
|---|---|
| Descripción | El sistema deberá permitir que el personaje sea alcanzado por habilidades lanzadas por otros usuarios y que le realicen daño basado en el ataque del lanzador y la defensa del receptor. |

*Tabla 5.8. RF-008. Impactar habilidades*

| RF-009 | Alteración de los atributos de los personajes |
|---|---|
| Descripción | El sistema deberá permitir que, bajo las circunstancias descritas en el juego, los atributos (vida, velocidad, ataque, defensa, etc) de los personajes se vean afectados tanto por habilidades como por objetos y cambien. |

*Tabla 5.9. RF-009. Alteración de los atributos de los personajes*

| RF-010 | Recogida de objetos en el suelo |
|---|---|
| Descripción | El sistema deberá permitir que los personajes recojan los objetos que salgan en el suelo del campo de batalla al pasar por encima de ellos y que, al hacer esto, realicen su funcionalidad. |

*Tabla 5.10. RF-010. Recogida de objetos en el suelo*

| RF-011 | Seleccionar kit de habilidades |
|---|---|
| Descripción | El sistema deberá permitir al usuario seleccionar el kit de habilidades con el que quiere jugar en la pantalla “Personalizar”. Éste debe quedar marcado. Los otros dos kits disponibles no tendrán esa marca. |

*Tabla 5.11. RF-011. Seleccionar kit de habilidades*

| RF-012 | Personalización de kit de habilidades |
|---|---|
| Descripción | El sistema deberá permitir al usuario personalizar las habilidades de cada kit en la pantalla “Personalizar”, eligiendo las que quiera. |

*Tabla 5.12. RF-012. Personalización de kit de habilidades*

| RF-013 | Identificación del usuario |
|---|---|
| Descripción | El sistema deberá permitir al usuario identificarse en su dispositivo para que le carguen los kits de habilidades configurados por él en conexiones anteriores. |

*Tabla 5.13. RF-013. Identificación del usuario*

### 5.3.2. Requisitos no funcionales

Los requisitos no funcionales [40] son restricciones que afectan a los servicios o funciones del sistema, por ejemplo, restricciones de tiempo, sobre estándares o procesos de desarrollo que se deben usar, lenguajes de programación, tecnologías o servidores específicos, etc. Un requisito no funcional puede ser más crítico o importante incluso que uno funcional, ya que, si un requisito funcional no se cumple, simplemente el sistema pierde funcionalidad, pero si uno importante no funcional no se cumple, puede inutilizar el sistema entero.

| RNF-001 | Permiso de almacenamiento del dispositivo |
|---|---|
| Descripción | El sistema requerirá acceso al almacenamiento del dispositivo para instalar la aplicación. |

*Tabla 5.14. RNF-001. Permiso de almacenamiento del dispositivo*

| RNF-002 | Acceso a internet por parte del dispositivo móvil |
|---|---|
| Descripción | El sistema requerirá que el dispositivo móvil esté conectado a internet para poder realizar sus funciones de red. |

*Tabla 5.15. RNF-002. Acceso a internet por parte del dispositivo móvil*

| RNF-003 | Almacenamiento mínimo |
|---|---|
| Descripción | El sistema requerirá de al menos 40MB de almacenamiento para poder instalarse. |

*Tabla 5.16. RNF-003. Almacenamiento mínimo*

| RNF-004 | Compatibilidad con dispositivos Android |
|---|---|
| Descripción | El sistema deberá poder instalarse y ejecutarse correctamente en todos los dispositivos con Android a partir de la versión 4.0. |

*Tabla 5.17. RNF-004. Compatibilidad con dispositivos Android*

| RNF-005 | Poca latencia |
|---|---|
| Descripción | El tiempo de conexión entre el dispositivo y el servidor debe ser mínimo (inferior a 70 ms) para poder disfrutar de una buena experiencia de juego. |

*Tabla 5.18. RNF-005. Poca latencia*

| RNF-006 | Motor de desarrollo Unity |
|---|---|
| Descripción | El sistema deberá implementarse usando Unity 3D como motor de desarrollo. |

*Tabla 5.19. RNF-006. Motor de desarrollo Unity*

| RNF-007 | Lenguaje de programación |
|---|---|
| Descripción | El código de programación del sistema deberá estar escrito en el lenguaje C#. |

*Tabla 5.20. RNF-007. Lenguaje de programación*

| RNF-008 | Proceso de desarrollo |
|---|---|
| Descripción | El proceso de desarrollo por el cual el proyecto se llevará a cabo será el Proceso Unificado. |

*Tabla 5.21. RNF-008. Proceso de desarrollo*

| RNF-009 | Facilidad de uso |
|---|---|
| Descripción | Los usuarios deberán poder utilizar las características básicas de juego después de 15 minutos de familiarizarse con él. |

*Tabla 5.22. RNF-009. Facilidad de uso*

## 5.3.3. Modelo de Casos de Uso

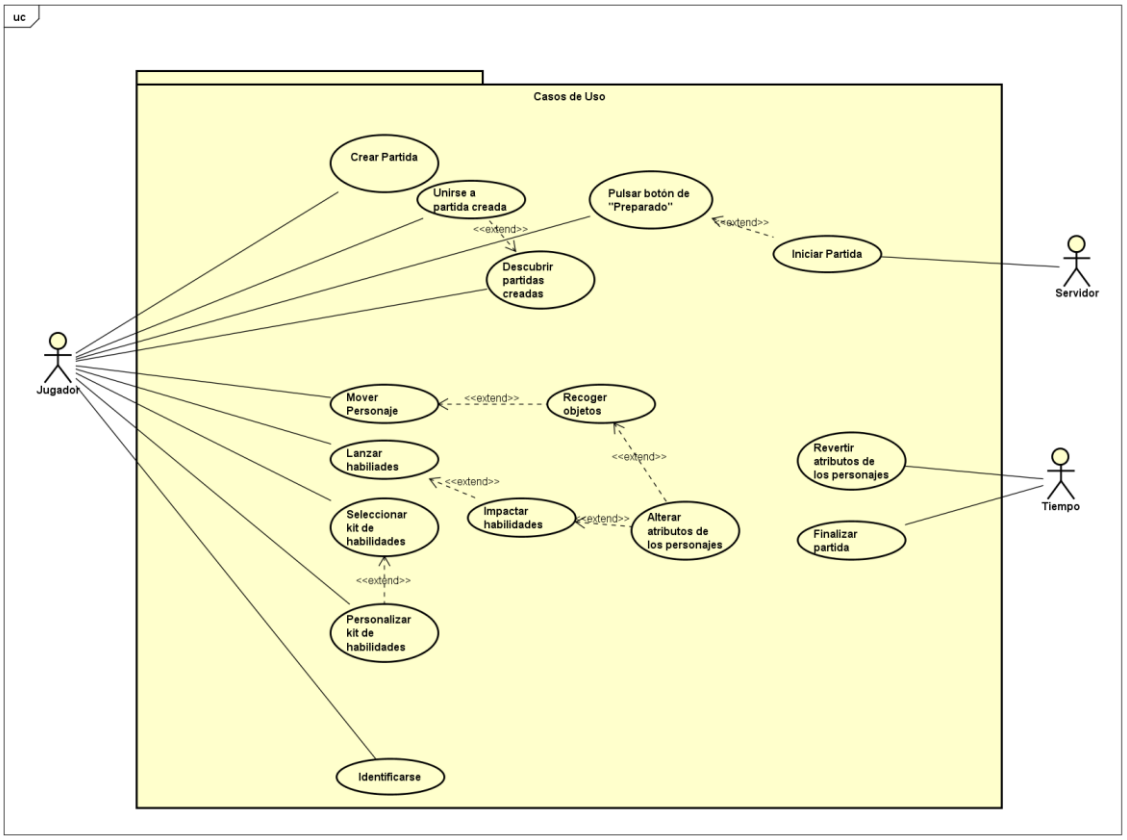


*Figura 5.1. Modelo de Casos de Uso*

#### 5.3.3.1. Actores de nuestro sistema

Un actor es cualquier entidad que interactúa con el sistema y que se relaciona con el mismo a través de los medios de interacción disponibles. En nuestro caso, habrá 3.

• **Jugador**:

Es el principal actor del sistema, es el encargado de crear las salas de las partidas, mover al personaje, lanzar habilidades y configurar las mismas.

• **Tiempo**:

El actor tiempo se encargará de todo lo relacionado con el tiempo dentro del juego. Por ejemplo, finalizar la partida automáticamente cuando se llegue al tiempo límite y hacer que los cambios de atributos de los personajes se reviertan pasada la duración del *PlayerModifier*.

• **Servidor**:

Es el encargado de realizar la gestión de las salas y la inicialización de las partidas.

#### 5.3.3.2. Descripción de los Casos de Uso

A continuación, desarrollamos los casos de uso para especificar un poco más de qué trata cada uno y cómo se ha pensado que debería ser la interacción con el sistema.

Cabe mencionar que todos los casos de uso en los que aparezca de alguna manera el servidor o dependan de la red, pueden experimentar secuencias de excepción que hagan que no siga ninguna de las secuencias recogidas aquí debidas al mal estado de la conexión. Como, por ejemplo, que cuando el servidor muestre las partidas disponibles, no muestre nada, aunque realmente sí tenga alguna activa. En cuanto a dentro de la partida, lo que pueda pasar por mal funcionamiento de la red puede ir desde desincronización de los elementos de la partida hasta directamente la desconexión de la partida.

#### 5.3.3.2.1. Mover Personaje

| | |
|---|---|
| **UC-001** | Mover personaje |
| **Actor** | Jugador |
| **Precondición** | El personaje tiene que estar vivo en el campo de batalla. |
| **Postcondición** | El personaje se ha movido en la dirección que el usuario ha indicado con el *joystick* si cumplía los requisitos para ese movimiento. |
| **Secuencia base** | 1. El usuario mueve el puntero del *joystick* izquierdo para indicar a donde quiere mover el personaje.<br>2. El sistema comprueba que el personaje cumple los requisitos para ese movimiento (no quiere avanzar a través de una piedra ni está "aturdido"). Si cumple los requisitos, el sistema desplaza al personaje una distancia proporcional a su velocidad. |
| **Secuencia alternativa** | 2b. El sistema comprueba si cumple los requisitos y no los cumple, por tanto, el personaje no se mueve del sitio. |
| **Comentarios adicionales** | |

*Tabla 5.23. UC-001. Mover Personaje*

#### 5.3.3.2.2. Lanzar habilidades

| | |
|---|---|
| **UC-002** | Lanzar habilidades |
| **Actor** | Jugador |
| **Precondición** | El personaje tiene que estar vivo en el campo de batalla. |
| **Postcondición** | El personaje ha lanzado la habilidad deseada con el *joystick* pertinente si cumplía los requisitos para ese lanzamiento. |
| **Secuencia base** | 1. El usuario mueve cualquiera el *joystick* de habilidades deseado hacia la dirección en la que quiere lanzar la habilidad.<br>2. El sistema comprueba si cumple los requisitos para lanzar la habilidad, si es así, se crea un proyectil de habilidad (*SkillshotProjectile*) y viaja por el campo durante una distancia determinada. |
| **Secuencia alternativa** | 2b. El sistema comprueba si cumple los requisitos para lanzar la habilidad. No los cumple, porque, por ejemplo, el enfriamiento de la habilidad aún está activo y tiene que esperar a que se reestablezca. Por tanto, ningún proyectil es creado. |
| **Comentarios adicionales** | |

*Tabla 5.24. UC-002. Lanzar habilidades*

#### 5.3.3.2.3. Impactar habilidades

| | |
|---|---|
| **UC-003** | Impactar habilidades |
| **Actor** | Jugador |
| **Precondición** | Un proyectil de habilidad ha sido lanzado y está viajando por el campo de batalla en una dirección. |
| **Postcondición** | Si se encuentra con algún jugador, que no haya sido el mismo lanzador, le afecta la habilidad impactada y realiza daño calculado entre el ataque del lanzador y la defensa del receptor. Si no encuentra ninguno, llegado el final de su rango, desaparece |
| **Secuencia base** | 1. El sistema continúa moviendo por el campo de batalla el proyectil hasta que golpea a un jugador. El caso de uso continúa en [Alterar atributos de los personajes]. |
| **Secuencia alternativa** | 1b. El sistema continúa moviendo por el campo de batalla el proyectil hasta que agota su distancia recorrida, conocida como "rango" y al ocurrir esto, el proyectil desaparece. |
| **Comentarios adicionales** | |

*Tabla 5.25. UC-003. Impactar habilidades*

#### 5.3.3.2.4. Recoger objetos

| | |
|---|---|
| **UC-004** | Recoger objetos |
| **Actor** | Jugador |
| **Precondición** | El personaje tiene que estar vivo en el campo de batalla y moviéndose. |
| **Postcondición** | El personaje ha pasado por encima de un objeto y lo ha recogido. |
| **Secuencia base** | 1. El usuario mueve al personaje hasta un objeto, hasta que pasa por encima de él.<br>2. El sistema procesa que ese personaje ha recogido ese objeto, lo hace desaparecer del campo y continúa por el caso de uso **Alterar atributos de los personajes**. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.26. UC-004. Recoger objetos*

#### 5.3.3.2.5. Alterar atributos de los personajes

| | |
|---|---|
| **UC-005** | Alterar atributos de los personajes |
| **Actor** | Jugador |
| **Precondición** | El personaje ha sido afectado por alguna de las situaciones en las que tiene lugar la alteración de sus atributos, como es el impacto por parte de una *SkillshotProjectile* o al recoger un objeto. |
| **Postcondición** | Los atributos pertinentes son alterados |
| **Secuencia base** | 1. El sistema reconoce qué modificador (*PlayerModifier*) es el que afecta al personaje y le altera los atributos pertinentes. También calcula junto con la duración del *PlayerModifier*, el momento en el tiempo en el que esa modificación se tiene que revertir (*tiempoFinalModifier*), lo que daría paso al caso de uso **Revertir atributos de los personajes**. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | Esta situación viene derivada de una acción de un jugador, por tanto, el actor es el jugador, aunque no participe activamente en este caso de uso. |

*Tabla 5.27. UC-005. Alterar atributos de los personajes*

#### 5.3.3.2.6. Revertir atributos de los personajes

| | |
|---|---|
| **UC-006** | Revertir atributos de los personajes |
| **Actor** | Tiempo |
| **Precondición** | El personaje ha sido afectado anteriormente por una alteración de atributos de personajes. |
| **Postcondición** | La alteración de atributos original se revierte. |
| **Secuencia base** | 1. Cuando llega el instante calculado anteriormente con esa alteración de atributos (*tiempoFinalModifier*), el sistema revierte los atributos específicos que se cambiaron en un inicio. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.28. UC-006. Revertir atributos de los personajes*

#### 5.3.3.2.7. Identificarse

| | |
|---|---|
| **UC-007** | Identificarse |
| **Actor** | Jugador |
| **Precondición** | Si se quiere acceder a los datos de un perfil ya creado, el usuario tiene que estar dado de alta en el sistema. |
| **Postcondición** | El usuario está identificado en el sistema. |
| **Secuencia base** | 1. El usuario introduce su nombre de usuario.<br>2. El sistema comprueba si el nombre de usuario existe. Si existe recupera de la base de datos los datos de las configuraciones de los kits del usuario. |
| **Secuencia alternativa** | 2b. El sistema comprueba si el nombre de usuario existe. Si no existe crea una configuración básica por defecto para que el nuevo usuario pueda configurar, ya con este nombre de usuario, sus kits de habilidades. |
| **Comentarios adicionales** | Este caso de uso sería como “registrarse” e “identificarse” en uno solo. Como no hay un registro como tal, ambas secuencias posibles están en este caso de uso. |

*Tabla 5.29. UC-007. Identificarse*

#### 5.3.3.2.8. Seleccionar kit de habilidades

| | |
|---|---|
| **UC-008** | Seleccionar kit de habilidades |
| **Actor** | Jugador |
| **Precondición** | El usuario tiene que estar previamente identificado con su nombre de usuario. |
| **Postcondición** | Se selecciona el kit de habilidades deseado y se va a la escena de personalización. |
| **Secuencia base** | 1. El usuario selecciona un kit de habilidades haciendo clic en él para personalizarlo.<br><br>2. El sistema redirige a la escena de personalización del kit seleccionado. |
| **Secuencia alternativa** | 1b. El usuario hace clic sobre la estrella a la derecha de cada kit de habilidades para marcar el deseado como favorito.<br><br>2b. El sistema marca como nuevo favorito el kit de habilidades elegido |
| **Comentarios adicionales** | |

*Tabla 5.30. UC-008. Seleccionar kit de habilidades*

#### 5.3.3.2.9. Personalizar kit de habilidades

| | |
|---|---|
| **UC-009** | Personalizar kit de habilidades |
| **Actor** | Jugador |
| **Precondición** | El usuario ha seleccionado previamente qué kit de habilidades quiere personalizar. |
| **Postcondición** | El kit de habilidades queda guardado de acuerdo con las preferencias del usuario. |
| **Secuencia base** | 1. El usuario pulsa sobre el nombre para editarlo.<br>2. El sistema permite al usuario introducir el nombre y lo guarda en el espacio destinado para el nombre.<br>3. El usuario selecciona una S*killshot* de las disponibles<br>4. El sistema muestra la información de la S*killshot* en pantalla<br>5. El usuario puede elegir escogerla como una de las deseadas pulsando seguidamente en una de las tres ranuras para colocarlas.<br>6. El sistema asigna la *Skillshot* a la ranura.<br>7. El usuario pulsa la pestaña "Pasivas"<br>8. El sistema oculta las *Skillshots* disponibles y en su lugar aparecen las pasivas disponibles.<br>9. El usuario selecciona una *pasiva* de las disponibles<br>10. El sistema muestra la información de la pasiva en pantalla<br>11. El usuario puede elegir escogerla como una de las deseadas pulsando seguidamente en una de las tres ranuras para colocarlas.<br>12. El sistema asigna la *pasiva* a la ranura.<br>13. El usuario pulsa el botón de Guardar.<br>14. El sistema guarda todos los cambios de ese kit de habilidades en el sistema de forma permanente. |
| **Secuencia alternativa** | 13b. En caso de que se realice cualquier otro cambio y se salga sin pulsar el botón de guardar, esos cambios no quedarán guardados. |
| **Comentarios adicionales** | El orden de edición de los distintos elementos personalizables dentro de esta escena no tiene que ser necesariamente como indica la secuencia base, al menos del paso 3 al paso 11, sino que pueden alternarse y repetirse tantas veces como considere el usuario, siempre y cuando el paso de guardar sea el último. |

*Tabla 5.31. UC-009. Personalizar kit de habilidades*

#### 5.3.3.2.10. Crear partida

| | |
|---|---|
| **UC-010** | Crear partida |
| **Actor** | Jugador |
| **Precondición** | El usuario tiene que estar previamente identificado con su nombre de usuario. |
| **Postcondición** | El usuario ha creado una partida en el servidor. |
| **Secuencia base** | 1. El jugador introduce el nombre de la partida, por el cual desea que sea buscada y pulsa "Crear partida"<br>2.El sistema envía la petición al servidor, si todo va bien, la sala de la partida se crea y el usuario es introducido en ella. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.32. UC-010. Crear partida*

#### 5.3.3.2.11. Descubrir partidas creadas

| | |
|---|---|
| **UC-011** | Descubrir partidas creadas |
| **Actor** | Jugador |
| **Precondición** | El usuario tiene que estar identificado en el sistema. |
| **Postcondición** | El usuario descubre todas las partidas que está almacenando el servidor en ese momento. |
| **Secuencia base** | 1. El usuario pulsa el botón "Descubrir partidas".<br>2. El sistema envía la petición al servidor y este responde con las partidas que tiene almacenadas y disponibles para unirse en ese momento. El sistema le muestra al usuario un listado de ellas. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.33. UC-011. Descubrir partidas creadas*

#### 5.3.3.2.12. Unirse a partida creada

| | |
|---|---|
| **UC-012** | Unirse a partida creada |
| **Actor** | Jugador |
| **Precondición** | El usuario debe haber realizado el caso de uso [Descubrir partidas creadas] con anterioridad |
| **Postcondición** | El usuario se une a la partida deseada. |
| **Secuencia base** | 1.El usuario identifica la partida a la que quiere unirse y pulsa el botón de "Unirse".<br>2.El sistema manda la petición al servidor y se incluye al usuario en la sala de la partida. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.34. UC-012. Unirse a partida creada*

#### 5.3.3.2.13. Pulsar botón de “preparado”

| | |
|---|---|
| **UC-013** | Pulsar botón de “preparado” |
| **Actor** | Jugador |
| **Precondición** | El jugador tiene que estar dentro de una sala de partida. |
| **Postcondición** | Si los demás usuarios también han pulsado el botón, se dará paso al caso de uso [Iniciar Partida]. |
| **Secuencia base** | 1. El usuario personaliza su nombre de partida y el color con el que quiere jugar y pulsa el botón de "Preparado" cuando lo está.<br>2. El sistema manda esa actualización al servidor para que se actualice en la sala de espera del resto de jugadores. Si éste era el último en estar preparado, se da paso al caso de uso [Iniciar Partida]. |
| **Secuencia alternativa** | 2b. El sistema manda esa actualización al servidor para que se actualice en la sala de espera del resto de jugadores. El jugador y el resto de ellos que ya hayan pulsado el botón deben esperar a que el último pulse el botón y así poder dar paso al caso de uso [Iniciar Partida]. |
| **Comentarios adicionales** | |

*Tabla 5.35. UC-013. Pulsar el botón de “preparado”*

#### 5.3.3.2.14. Iniciar partida

| | |
|---|---|
| **UC-014** | Iniciar partida |
| **Actor** | Servidor |
| **Precondición** | Hay un mínimo de jugadores y todos están "preparados" para jugar en una sala de partida. |
| **Postcondición** | La partida se inicia y permite jugar a todos los jugadores. |
| **Secuencia base** | 1. El servidor ha recibido la última petición de "preparado". Inicia una cuenta atrás y se la notifica al sistema.<br>2. El sistema notifica la cuenta atrás a los usuarios.<br>3. El servidor carga a cada jugador en una posición distinta del mapa y se lo envía al sistema.<br>4. El sistema de cada jugador le carga la información necesaria para jugar la partida (cargar el mapa) y colocar al resto de los jugadores. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.36. UC-014. Iniciar partida*

#### 5.3.3.2.15. Finalizar partida

| | |
|---|---|
| **UC-015** | Finalizar partida |
| **Actor** | Tiempo |
| **Precondición** | La partida se encuentra en curso. |
| **Postcondición** | La partida queda finalizada. |
| **Secuencia base** | 1. Periódicamente, se comprueba si se ha llegado ya al tiempo límite de partida, si es así se le notifica a los usuarios y la partida termina. |
| **Secuencia alternativa** | |
| **Comentarios adicionales** | |

*Tabla 5.37. UC-015. Finalizar partida*

## 5.4. Flujo de actividad del jugador

En este apartado se presenta un diagrama de actividad realizado para conceptualizar, con un grado de detalle bastante más alto que los casos de uso, el flujo de actividades que tendrá el personaje del jugador dentro de la partida del videojuego, para ayudar a comprender la idea de su funcionamiento y así facilitar el diseño y la implementación.

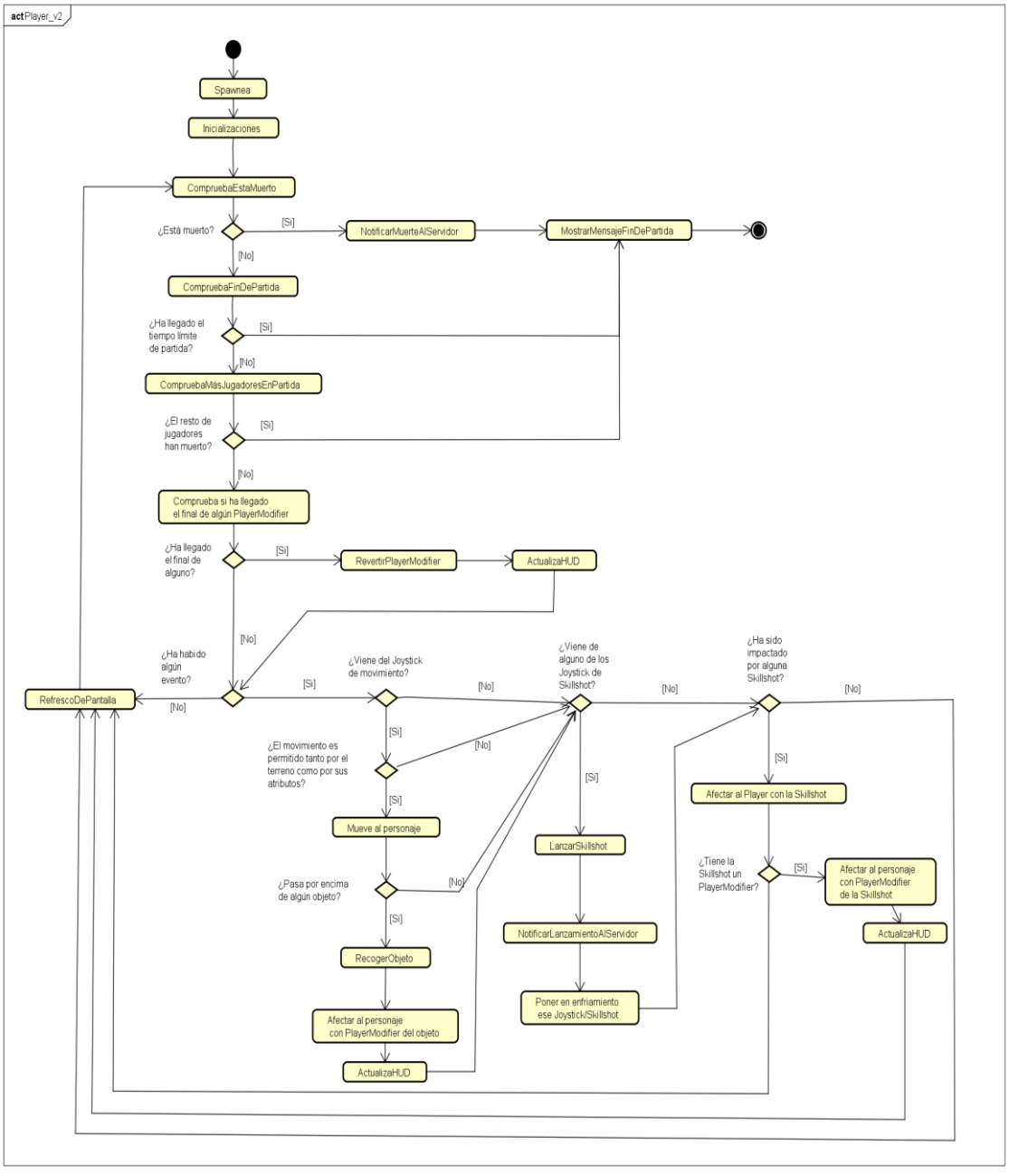


*Figura 5.2. Diagrama de actividad de Player*

# 6. Diseño

En este apartado entraremos en detalle sobre cómo llevar a cabo la solución de los casos de uso descritos en el apartado anterior. En los siguientes apartados explicamos en detalle los dos modelos de dominio que van a ser importantes para la aplicación.

El primero que explicamos será el que nos brinda Unity. El motivo de esto es que, como es nuestro motor de videojuego, vamos a tener que aplicar sus clases a nuestra solución. El diagrama concreto para la aplicación se apoya en conceptos explicados en el de Unity, por eso es conveniente presentarlo primero.

Una vez vistos los modelos de dominio, en el apartado de patrones de diseño, veremos cómo afrontaremos algunos problemas con ayuda de patrones, tanto los que Unity, indirectamente, nos recomienda (o en ocasiones obliga a) usar, como los elegidos por el equipo para cada situación en particular.

## 6.1. Modelo de dominio de Unity

Al usar Unity como entorno de desarrollo y ayudarnos de su amplia API para desarrollar, conviene explorar primero qué es lo que ya nos ofrece. En la siguiente figura tenemos un diagrama explicativo de la estructura de las clases más importantes de Unity. No aparecen todas, pero es suficiente para el objetivo deseado.

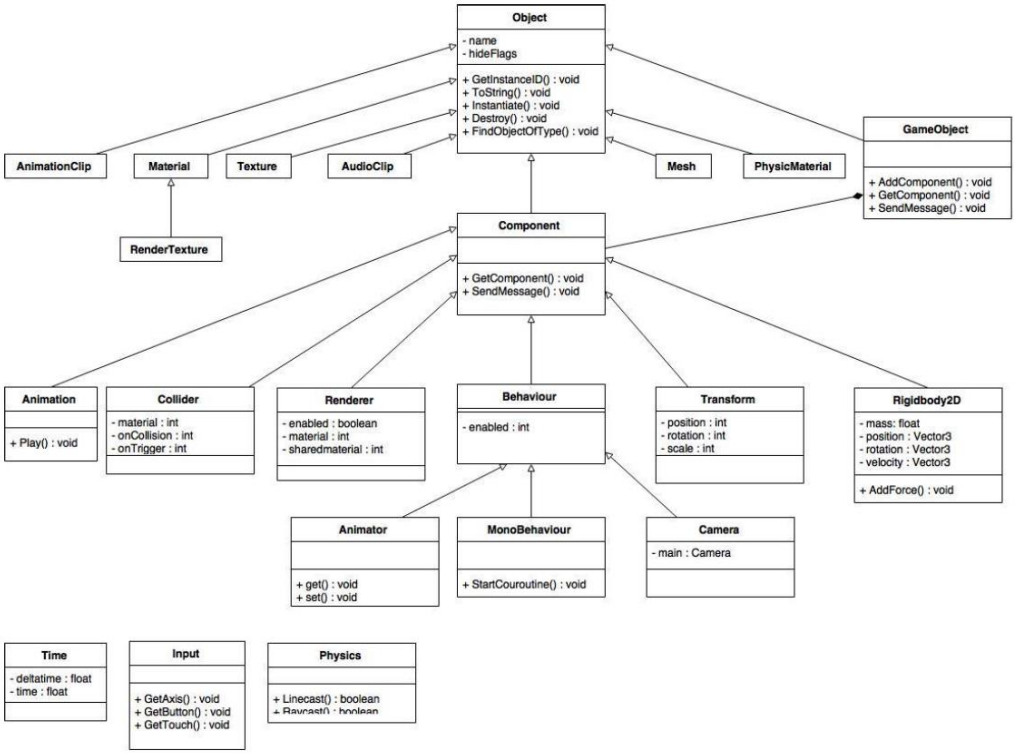


*Figura 6.1. Modelo de dominio de Unity Fuente: [41]*

### 6.1.1. Explicación del modelo de dominio de Unity

Viendo el modelo de dominio con las clases principales que nos brinda Unity, vamos a pararnos a explicar algunas de las clases más importantes y destacables para nuestro proyecto. Para entender mejor algunos conceptos referentes a la explicación, es importante saber que Unity trabaja con “Escenas”.

Las escenas son espacios vacíos en un principio. Solamente disponen de una cámara, la cual es vital para que el jugador pueda ver todos los elementos que se irán añadiendo a la escena. Algunos de estos elementos serán los objetos o incluso combinaciones de los objetos cuyas clases explicaremos a continuación.

#### 6.1.1.1. Object

Clase raíz de la que heredan el resto de clases del modelo de dominio de Unity. Tiene algunas de las funciones más básicas y a la vez, más importantes y utilizadas, como son por ejemplo *Instantiate*, *GetInstanceID* y *Destroy.*

La primera, sirve para instanciar cualquier objeto dentro de nuestra escena, clonando un objeto original y retornando el clon.

El método *GetInstanceID* devuelve el número interno de la instancia de cada objeto. Muy útil a la hora de detectar errores y usar el *debugger*.

*Destroy* elimina un *GameObject*, un *Component* o un *asset* de la escena.

#### 6.1.1.2. GameObject

Es la clase base para todas las entidades que vayan a aparecer en las escenas. Dispone de varios atributos y funciones que ayudan al manejo de estos objetos y sus componentes. Básicamente, un *GameObject* es un contenedor de Componentes, los cuales le darán vida y le harán diferenciarse del resto de *GameObject*. Entre sus funciones más importantes tenemos AddComponent, GetComponent*, GetComponentInChildren, SetActive* y *Find.*

• **AddComponent** sirve para añadir algún componente de manera dinámica en el script, aunque la mayoría de los componentes de un *GameObject* se van a añadir desde el editor de Unity.

• **GetComponent** y **GetComponentInChildren** son usados para buscar y recoger los componentes del tipo deseado que se encuentren, bien en el *GameObject* o bien en los hijos (dentro de la jerarquía) de ése *GameObject*.

• **SetActive** es una función que podemos utilizar tanto para activar como para desactivar los *GameObject* dentro de la escena. Si se desactiva un *GameObject*, todos sus hijos quedarán desactivados también.

• **Find** es una función que encuentra *GameObjects* dentro de la escena actual con el mismo nombre que se pasa por parámetro. Útil para manejar desde script elementos creados desde el editor de escenas de Unity.

#### 6.1.1.3. Component

Es la clase base para todas las entidades que van a ser contenidas por un *GameObject*. Son los que le van a dar vida al *GameObject*. Explicaremos algunos de los componentes más importantes, todos los que mencionaremos heredan directamente de la clase *Component*.

#### 6.1.1.4. Transform

Describe la posición, rotación y escala del objeto. Todos los objetos de la escena deben tener un *transform*. Al manejar estas tres características, también se manejan las de sus hijos, siguiendo la jerarquía. De manera que, si mueves arriba a la derecha al componente padre, toda la jerarquía de hijos se moverá a partir de ese *transform* del padre. Funciona igual con las otras dos características. Entre las funciones más destacables tenemos *Rotate*, *Translate*.

Estas sirven para rotar y trasladar el objeto los grados y dirección, respectivamente, de los objetos.

#### 6.1.1.5. Renderer

Un *Renderer* es el que hace que los objetos aparezcan y sean visibles en la escena. Entre los tipos de *Renderer*, el que más vamos a utilizar es el *SpriteRenderer*.

#### 6.1.1.6. Collider

Los *colliders* permiten definir las superficies de colisión de los objetos de las escenas. Se pueden crear *collider* con muchas formas, por ejemplo, círculos, triángulos, cuadrado, etc. El que más usaremos será *Box Collider 2D*.

#### 6.1.1.7. Behaviour

Los *behaviours* son componentes que se pueden activar y desactivar. El que vamos a utilizar más, y por tanto desarrollamos a continuación es *MonoBehaviour*, que, debido a su importancia, tiene un apartado al nivel de *Component*.

#### 6.1.1.8. MonoBehaviour

*MonoBehaviour* es la clase base de la cual todos los scripts de Unity heredan. Cuando usemos C# para los *scripts*, hay que heredar explícitamente de MonoBehaviour. Al crear un script y heredar de *MonoBehaviour* lo que hacemos es darle un comportamiento en concreto a un *GameObject*, al cual le asignaremos este *script* (*MonoBehaviour*) como un componente más. Dentro de este script podemos definir lo que nuestro *GameObject* hará dentro de la escena, controlando el resto de sus componentes con nuestro código. Al crear un script desde el editor, automáticamente se nos genera con varias funciones, pertenecientes a MonoBehaviour. Éstas son:

##### 6.1.1.8.1. Start

*Start* es una función que se llama en el primer *frame* (fotograma) en que el *script* es activado, justo antes de cualquiera de las llamadas a *Update*. Se suele usar, como *Awake*, para la inicialización de variables.

#### 6.1.1.8.2. Update

*Update* es una función que es llamada cada *frame*, si el *MonoBehaviour* está activo. *Update* es la función más utilizada para implementar cualquier tipo de comportamiento. Por ejemplo, si queremos hacer andar a un personaje, debemos hacer una pequeña traslación en cada *frame*, para dar al jugador la sensación de que, en cada fotograma, el personaje avanza poco a poco.

Existen otras funciones muy útiles, que, aunque no sean las dos principales, pueden proporcionar otras posibilidades que las dos funciones mencionadas anteriormente no pueden dar. Son las siguientes.

#### 6.1.1.8.3. Awake

*Awake* se llama cuando la instancia del *script* está siendo cargada. Solamente se llama una vez en el ciclo de vida de la instancia de ese *script*, justo al principio. Es una función que se ejecuta justo antes que *Start* y te asegura que se va a ejecutar siempre antes que el *Start* de cualquier otro GameObject de la escena. Esto puede ser muy útil para localizar objetos en la escena sin miedo a que algún *Start* de otro objeto se haya ejecutado y que esto pueda provocar un error. Se suele usar, como *Start*, para la inicialización de variables.

#### 6.1.1.8.4. FixedUpdate

Con un comportamiento similar a *Update* pero con la singularidad de que tiene la frecuencia del sistema de físicas. Es llamada a un ritmo constante, cada 0.02 segundos (50 llamadas por segundo) que es el tiempo por defecto entre llamadas. Para acceder a este valor se puede con *Time.fixedDeltaTime*.

Una especialización de *MonoBehaviour* que no aparece en el diagrama anterior pero sí vamos a utilizar es *NetworkBehaviour*, orientado al uso de redes en los juegos. Contiene multitud de variables y métodos que son muy útiles para el manejo y gestión de la partida en red. Hablaremos un poco más en detalle de lo que nos ofrece en el apartado de implementación.

## 6.2. Modelo de dominio

En este apartado veremos, en primer lugar, un modelo de dominio que solamente representa las clases que realmente son “importantes” y representan al dominio de la cuestión. Todas tienen una función bien definida y son vitales para el desarrollo del videojuego. Más adelante mostraremos un diagrama ampliado con la totalidad de las clases que participan en el correcto funcionamiento del proyecto.

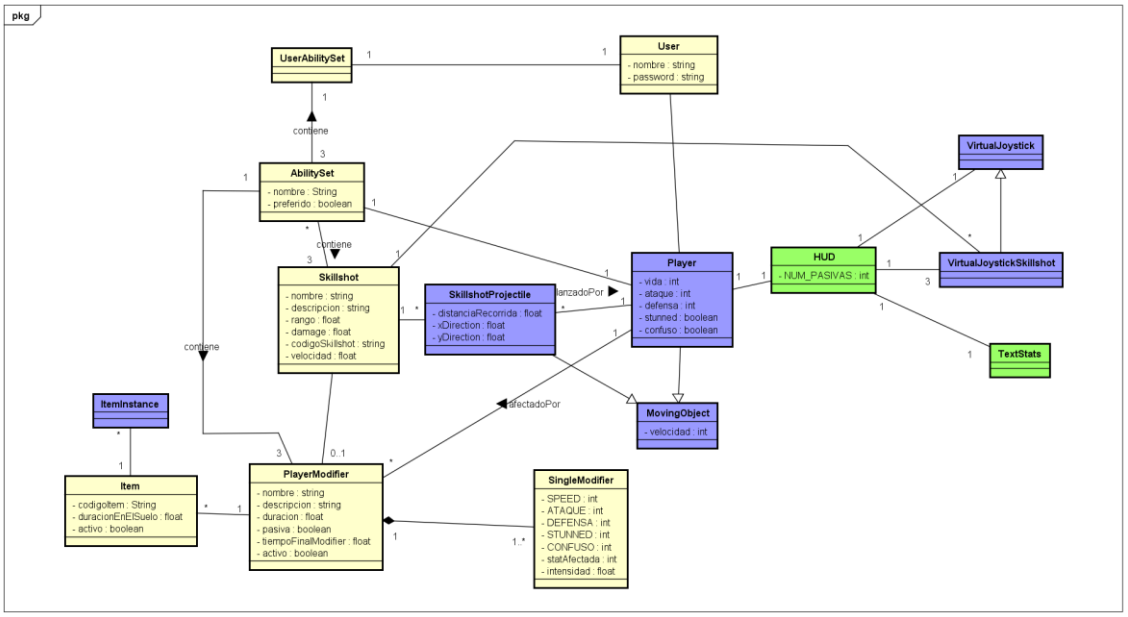


*Figura 6.2. Modelo de dominio del proyecto*

Con el objetivo de evitar la redundancia y repetición a la hora de hablar de cada una de las clases, en el diagrama se colorearán las clases para diferenciarlas. La diferenciación radica en la clase de la que hereda cada una. Es preferible hablar de las particularidades de cada una en su apartado y hablar de las herencias y el porqué de ellas, como grupo, aquí.

En primer lugar, hablaremos de las coloreadas en amarillo, que, son clases, que no heredan de ninguna clase más que de la raíz. Esto se debe a que no necesitamos que estas clases tengan un comportamiento de un *Behaviour* o de un *Component*, quizás ni siquiera queremos que ese elemento aparezca en la escena, simplemente necesitamos una clase sencilla para albergar atributos y funciones muy específicas que nada tienen que ver con los elementos mencionados anteriormente. Es el caso, por ejemplo, de las *Skillshots* y los *PlayerModifier*, entre otros. ¿Por qué necesitamos que estas clases sean simples? Porque las vamos a usar como una clase normal. Solo queremos consultar sus atributos y ejecutar sus funciones, pero no necesitamos que, por ejemplo, ocurra algo cada vez que se recarga la pantalla (funciones *Update* y

*FixedUpdate*). Las tendremos como “almacenadores de información” muy valiosa para el juego, pero no como elementos de la escena como tal.

El siguiente grupo son los coloreados de verde. Estas clases son las que heredan de *MonoBehaviour*. Son clases que, a diferencia de las anteriores sí vamos a querer que tengan un espacio en nuestras escenas, porque va a ser necesario que en cada refresco de pantalla realicen alguna acción, o que simplemente vayan a aparecer en la escena. El ejemplo más claro es el HUD.

Por último, el grupo morado. Estas clases son las que heredan de *NetworkBehaviour*. Las queremos básicamente para lo mismo que las anteriores, las del grupo verde, con la particularidad de que estas clases van a ser usadas expresamente en red. Debido a esto, van a tener que heredar de *NetworkBehaviour* porque les provee de ciertos métodos y variables que van a ser muy necesarios para desarrollar sus actividades en red. Los *VirtualJoystick* también van a heredar de esta clase, aunque a priori pueda parecer que no es necesario ya que “forman parte” del HUD. Esto es debido a que no solamente es necesario para las clases que vayan a usar explícitamente los métodos que brinda *NetworkBehaviour*, sino que, en ocasiones, como el caso que mencionamos, es necesario que lo hereden porque van a ser importantes en la red y necesitan tener un componente llamado *NetworkIdentity* (identificación en red). Sin que hereden de *NetworkBehaviour* no se puede añadir este componente.

### 6.2.1. Clases del modelo de dominio

#### 6.2.1.1. User

La clase en la cual estará la información del usuario, con la que el usuario podrá identificarse en el servidor y en el que se guardarán sus progresos.

#### 6.2.1.2. UserAbilitySet

Esta clase sirve para albergar un conjunto, en principio 3, de varios AbilitySet pertenecientes al usuario (User) que esté jugando.

#### 6.2.1.3. AbilitySet

Esta clase, como su nombre indica, va a ser el “Set de habilidades”. Consistirá en un conjunto de 3 Skillshots y 3 PlayerModifier que el usuario podrá personalizar a su gusto con los disponibles para escoger. También podrá marcar

uno de los 3 de los que dispone como favorito (es con el que jugará cuando decida iniciar la partida).

#### 6.2.1.4. PlayerModifier

Esta clase, como su nombre indica contendrá los “Modificadores del jugador”. Sus atributos serán la duración que tenga este modificador, nombre, descripción y, además, un conjunto de *SingleModifier*.

#### 6.2.1.5. SingleModifier

Esta clase contendrá simplemente la estadística a modificar del jugador y la intensidad con la que esta modificación se produce (una modificación muy fuerte o una más leve). Cada *PlayerModifier* tiene un conjunto de *SingleModifiers*. Cuando a un Player le afecta el *PlayerModifier*, se van afectando todos los *SingleModifier* que le contienen.

#### 6.2.1.6. Item

Clase que describe a un “objeto” que aparecerá de manera aleatoria en el suelo. Contiene información sobre su *codigoItem*, útil para saber qué imagen le corresponde. Además de, lo más importante, un *PlayerModifier* asociado que describirá los efectos que tiene sobre los jugadores al recoger este objeto del suelo.

#### 6.2.1.7. ItemInstance

Es la representación “física” del objeto en la escena. Contendrá la información sobre el ítem al que representa.

#### 6.2.1.8. Skillshot

Clase que describe los “ataques” o “disparos” dentro del juego. Tendrá diversos atributos como nombre, descripción, *damage* (daño), rango (rango máximo al que llegará el disparo antes de desaparecer sin golpear a nadie), velocidad, *cooldown* (enfriamiento para reutilizarse), etc. Todos estos atributos conforman una Skillshot, la diferenciación de los distintos atributos hace que cada *Skillshot* sea distinta. Así mismo se dispondrá de un “códigoSkillshot” el cual nos será útil

para identificar la imagen que se usará para diferenciarse de otros *Skillshot*, entre otros usos.

#### 6.2.1.9. MovingObject

Clase abstracta que describe el funcionamiento y contiene algo de implementación común de los objetos que serán "movibles" dentro del juego, que son los que describiremos a continuación.

#### 6.2.1.10. SkillshotProjectile

Clase que hereda de *MovingObject*, por tanto, objeto movible del juego. Lo que representa esta clase son los proyectiles de las *Skillshot*. Las *Skillshots* solamente describían sus atributos, pero los *SkillshotProjectile* son los proyectiles que el jugador va a poder ver cómo impactan sobre él o cómo impacta con ellos a sus rivales.

#### 6.2.1.11. Player

Clase de manejo del jugador o "personaje". Obviamente será un heredado de *MovingObject* ya que el jugador va a poder mover al personaje por el campo para intentar esquivar los *SkillshotProjectile* y para recoger los objetos. Contendrá mucho código referente al movimiento, a la trata (afectar y revertir) *PlayerModifier*, afectar *SkillshotProjectile* recibidas, etc. Entre sus atributos tendremos la vida, la vida máxima, el ataque, la defensa, la velocidad (heredada), etc.

#### 6.2.1.12. HUD

Esta clase manejará el HUD, la interfaz de la que dispondrá el jugador para saber la información referente a su personaje en todo momento. Para ayudar a esto, el HUD se compondrá de varios elementos, algunos de ellos los describiremos a continuación, otros, al carecer de importancia o no ser una clase nueva implementada, los obviaremos, ya que solo son para mejorar y hacer más intuitiva la experiencia de juego del usuario.

#### 6.2.1.13. VirtualJoystick

Clase que manejará el *joystick* situado a la parte izquierda de la pantalla, el cual manipulará el movimiento del personaje. Forma parte del HUD.

#### 6.2.1.14. VirtualJoystickSkillshot

Clase que hereda de *VirtualJoystick*, ya que comparten mucha implementación y su forma básica de funcionar es prácticamente la misma, con la diferencia de que *VirtualJoystickSkillshot* contendrá la imagen de una *Skillshot* y al soltar el cursor con una dirección, el personaje lanzará la *Skillshot* asociada a este objeto. Además de esto, al contener cada uno una *Skillshot* y usarla cuando se use el *joystick*, este entrará en "enfriamiento" y no se podrá lanzar más esa habilidad hasta que no se reestablezca. Forma parte del HUD.

#### 6.2.1.15. TextStats

Sencilla clase que solo se utilizará para mantener actualizadas las estadísticas y atributos del personaje para mostrar la información en todo momento a través del HUD.

## 6.3. Arquitectura lógica

La elección de la arquitectura [42] de un producto software tan complejo y extenso como es un videojuego puede resultar difícil. Dependiendo de en qué nos centremos, podemos pensar que el patrón arquitectónico puede ser *Modelo-Vista-Controlador.* Y así podría ser si solamente nos fijáramos en la parte de personalización y menús, pero la principal funcionalidad de este proyecto reside en la partida y la jugabilidad, que mucho distan de un *Modelo-Vista-Controlador*. Hay un patrón arquitectónico que se ajusta más a este sistema de eventos por los cuales el usuario va a realizar cambios en el sistema. Este patrón es la *Arquitectura Dirigida por Eventos (EDA)*. Esta arquitectura se centra en la creación, detección, consumo y reacción de eventos (por ejemplo, en Unity tenemos la clase *EventSystem* [43]). Nos referimos a evento como cualquier acción, venga del usuario o no que provoque un cambio significativo en un estado de algún elemento del sistema.

La arquitectura EDA consta de cuatro capas lógicas, cada una con su función definida que explicamos a continuación.

1. **Generador de Eventos.** En esta capa se detectan las apariciones de un hecho y genera el evento relacionado con ese hecho en consecuencia. Después transmiten ese evento al Canal de Eventos.
2. **Canal de Eventos.** Esta capa consiste en un mecanismo mediante el cual la información que le ha llegado desde un generador de eventos se transfiere al Motor de Procesamiento de Eventos.
3. **Motor de Procesamiento de Eventos.** El motor de esta capa es el encargado de identificar un evento y asignarle la respuesta que corresponda.
4. **Respuesta al Evento.** Es la parte final de la captura del evento. Esta capa muestra los resultados de la aparición del evento.

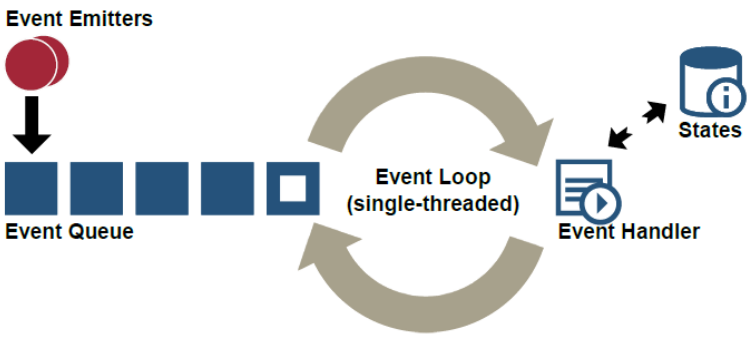


*Figura 6.3. Arquitectura Dirigida por Eventos*

Esta última capa representa la respuesta final que queremos que tenga este evento concreto, en nuestro caso los menús con botones tendrán su acción indicada en forma de función de una clase (como explicamos más adelante) y los *joysticks* realizan sus acciones mediante los métodos que se explican en el apartado del capítulo de Implementación relativo al movimiento.

## 6.4. Patrones de diseño

Christopher Alexander [44] reconocido arquitecto, define en los patrones de la siguiente manera.

> "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo siquiera dos veces de la misma forma".

ALEXANDER, C., ET AL.A PATTERN LANGUAGE: TOWNS, BUILDINGS, CONSTRUCTION, OXFORD UNIVERSITY PRESS, (1977).

Llevado al campo de la informática y, en concreto, del desarrollo, los patrones de diseño [45] son realmente útiles para afrontar problemas que surgen a la hora de diseñar un sistema. Nos ayudan a ver cómo, gente que ha tenido anteriormente los mismos problemas que nosotros, han consensuado que una forma más o menos concreta de solucionarlo es una de las mejores maneras para afrontarlos.

### 6.4.1. Patrones de diseño propuestos por Unity

Al trabajar con un entorno de desarrollo como Unity, con una API propia para ayudarnos a trabajar con los elementos del juego, es normal encontrar que propicie el uso de determinados patrones de diseño, a parte de los que el diseñador encuentre necesario utilizar para el juego en concreto.

#### 6.4.1.1. Flyweight

El patrón *Flyweight*, también conocido como objeto ligero, sirve para reducir la redundancia de información idéntica en objetos que van a aparecer en la escena en gran número.

De esta manera, si tenemos, por ejemplo, cientos de baldosas iguales en el suelo, la información idéntica que tengan entre sí, se aprovecha, no ocupando la misma memoria todos y cada uno como si fueran objetos totalmente distintos. Toda la información como color, tamaño y lo que compartan, es la misma, y simplemente ocuparán más memoria los atributos que sí distingan cada una entre sí, como por ejemplo la posición exacta de cada una.

#### 6.4.1.2. State

El patrón *State*, o Estado, se utiliza cuando el distinto comportamiento de un objeto depende del estado de este.

El patrón es usado para las máquinas de estados que sirven para gestionar las animaciones, que veremos más adelante. Simplemente dependiendo de la dirección en la que se esté moviendo el personaje, le diremos que está en un estado o en otro (por ejemplo, arriba a la izquierda o a la derecha). Entonces, cada objeto animación, dependiendo del estado en el que se encuentre el personaje, actuará a su manera.

### 6.4.2. Patrones de diseño elegidos para el proyecto

#### 6.4.2.1. Singleton

El patrón *Singleton*, o instancia única, es un patrón que obliga a que solo pueda haber un objeto de esa clase instanciado a la vez. Esto tiene muchas ventajas, la principal y más importante es que permite el acceso global a dicho objeto en todo momento y desde cualquier sitio.

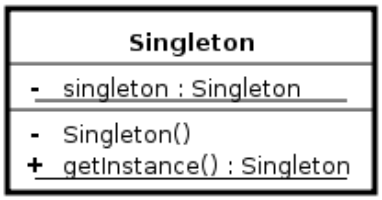


*Figura 6.4. Estructura básica de clase aplicando Singleton*

Como podemos ver, para obtener la instancia, se llama a un método de clase (la forma de acceder a ello de forma global) y ese método nos da la instancia que está creada en ese momento y que mantiene todos sus atributos.

Hemos pensado que para nuestra aplicación sería muy útil tener una clase que albergara toda la información que tiene que estar activa y accesible mientras el juego esté activo. Por consiguiente, la utilización del *Singleton* va a centrarse en una clase llamada *GameManager*. Esta clase se creará desde el inicio del juego y permanecerá con una instancia desde entonces hasta que lo cerremos, albergando en su interior todos los recursos que puedan ser necesarios para acceder de manera global a ellos. Podremos acceder desde las pantallas de personalización, guardando configuración en ella, así como desde la pantalla de juego.

### 6.4.2.2. Comando

Según describe la *Gang of Four* [46], autores del libro "Design Patterns: Elements of Reusable Object-Oriented Software", el patrón Comando es "Encapsular una solicitud como un objeto, permitiendo a los usuarios parametrizar clientes con diferentes solicitudes, solicitudes de cola o registro, y soporte de operaciones que se pueden deshacer". Básicamente, el objetivo es encapsular órdenes, solicitudes o comandos en objetos. Un uso adicional y optativo del patrón Comando consiste en que esas solicitudes puedes tener la opción de deshacerse.

En nuestro caso, hemos decidido aplicar una versión adaptada del patrón Comando en el diseño de cómo van a funcionar los *PlayerModifier*. Los *PlayerModifier* están considerados como unas "solicitudes" en las que varios atributos del personaje se alteran. Del patrón cumplen que estas solicitudes están encapsuladas en un objeto (*PlayerModifier*) y que tienen forma de aplicarse y de revertirse.

### 6.4.2.3. Observador

El patrón Observador es un patrón que define una dependencia de uno a varios entre distintos elementos, de manera que cuando ese "uno" cambia su estado, éste notifica a todos estos "varios" de sus cambios.

Hemos pensado que este patrón era el ideal para el HUD, debido a que tenemos elementos en éste que son observadores, y que deben "estar pendientes" y reflejar la información en cada momento de un mismo elemento, el *Player*.

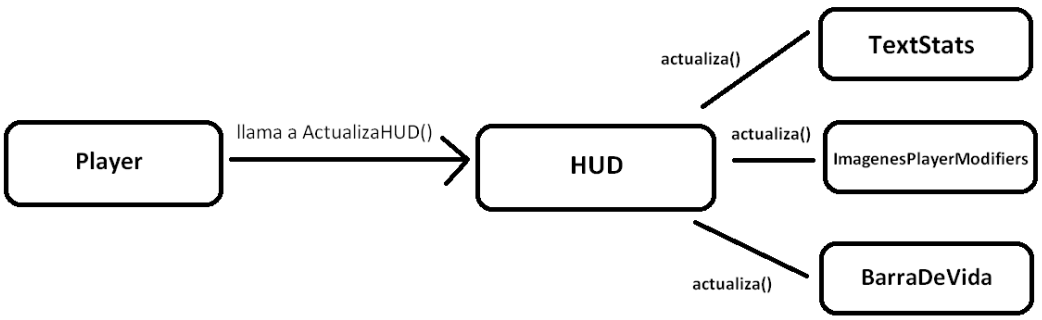


*Figura 6.5. Esquema explicativo del patrón observador en el caso concreto*

Como vemos en la figura, Player dispone de un método en su HUD asociado con el que puede notificar a todos sus observadores a la vez. El HUD centraliza a todos sus componentes y les notifica uno a uno que su Player ha cambiado algo.

Así todos y cada uno de los elementos pueden actualizar la información que le están mostrando al usuario.

#### 6.4.2.4. Fachada

La motivación del patrón Fachada es la necesidad de estructurar los desarrollos y reducir su complejidad mediante la división en subsistemas, haciendo que las comunicaciones entre jerarquías o paquetes sean las mínimas posibles y que éstas estén centralizadas.

En nuestro caso, vamos a utilizar el patrón Fachada para todas las acciones que el sistema requiera con la base de datos. Toda interacción con base de datos se realizará a través de la clase *SQLiteManager*, una clase propia. Su funcionamiento, utilizando *SQLite*, consiste en iniciar la base de datos, cerrarla, crear tablas, guardar información, recoger información, etc. Más adelante mencionamos algo más sobre esta clase.

#### 6.4.2.5. Adaptador

El patrón Adaptador se utiliza para adaptar clases con el objetivo de que puedan utilizar una interfaz de la que, en un principio, no pueden hacer uso. Para esto, se crea una interfaz que “adapte” la clase a la interfaz primera que quería usar.

Llevado a nuestro videojuego, utilizamos varias veces la idea del patrón. El caso más claro es el que tenemos con *UserAbilitySet* y *AbilitySet*. Estas son dos clases, que, como vimos anteriormente no heredaban de nada. Son clases simples, con sus atributos y funciones, pero simples.

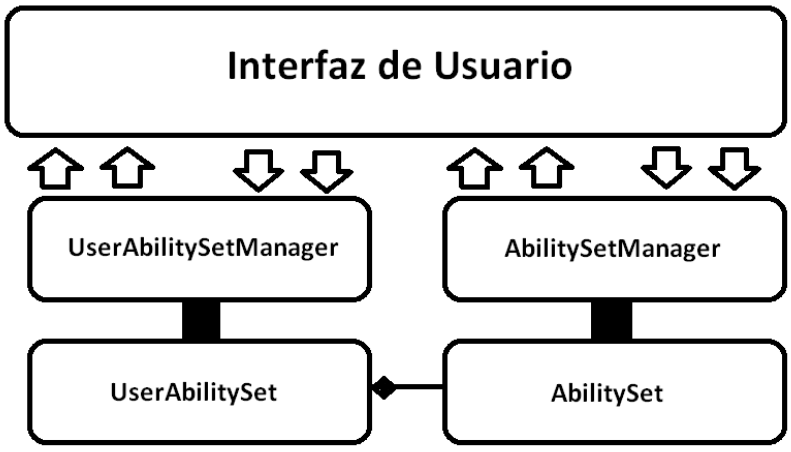


*Figura 6.6. Esquema de las clases "managers"*

Para poder manejar estos dos importantes elementos de manera correcta desde la interfaz de usuario (UI), hemos necesitado crear dos clases que adapten estas

clases simples al uso de UI e interacción con el usuario. Estas clases son *UserAbilitySetManager* y *AbilitySetManager*. Es la manera que tenemos para simular que el usuario está "tocando directamente" estas dos clases simples.

Otro uso que le hemos dado al patrón Adaptador ha sido los *Items*. Como pasa con el caso anterior, *Item* es una clase sencilla, con sus atributos y sus métodos, pero que no está preparada para "saltar al mundo de la arena". Por eso, para que pueda interactuar bien con los elementos de la arena, como por ejemplo un Player, se ha creado una clase que adapta a los *Items* para que puedan ser un elemento más de la arena, esta clase es *ItemInstance*.

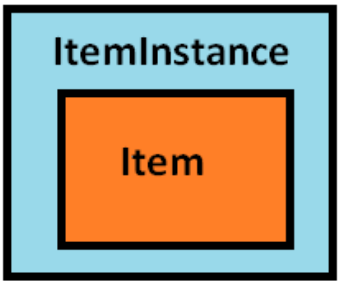


*Figura 6.7. Esquema de ItemInstance e Item*

Como vemos en la imagen, *ItemInstance* contiene en su interior un *Item*, o, mejor dicho, una referencia a éste. Dentro de la zona azul están todas las funciones y componentes de Unity necesarios para que el *Item* pueda ser uno más de la arena.

Para finalizar este apartado, el último uso que le hemos dado al patrón Adaptador ha sido el de las *Skillshot*. Ocurre algo muy parecido a lo que ocurre con *Item* e *ItemInstance*. La clase que adapta al *Skillshot* en este caso es el *SkillshotProjectile*. En el caso del *Item*, el *ItemInstance* apenas tiene funcionalidad aparte de hacer de intermediario y adaptarlo al medio. En cambio, en el caso de *SkillshotProjectile*, además de todo esto, implementa funcionalidades extra que veremos en el apartado de Implementación.

## 6.5. Modelo relacional

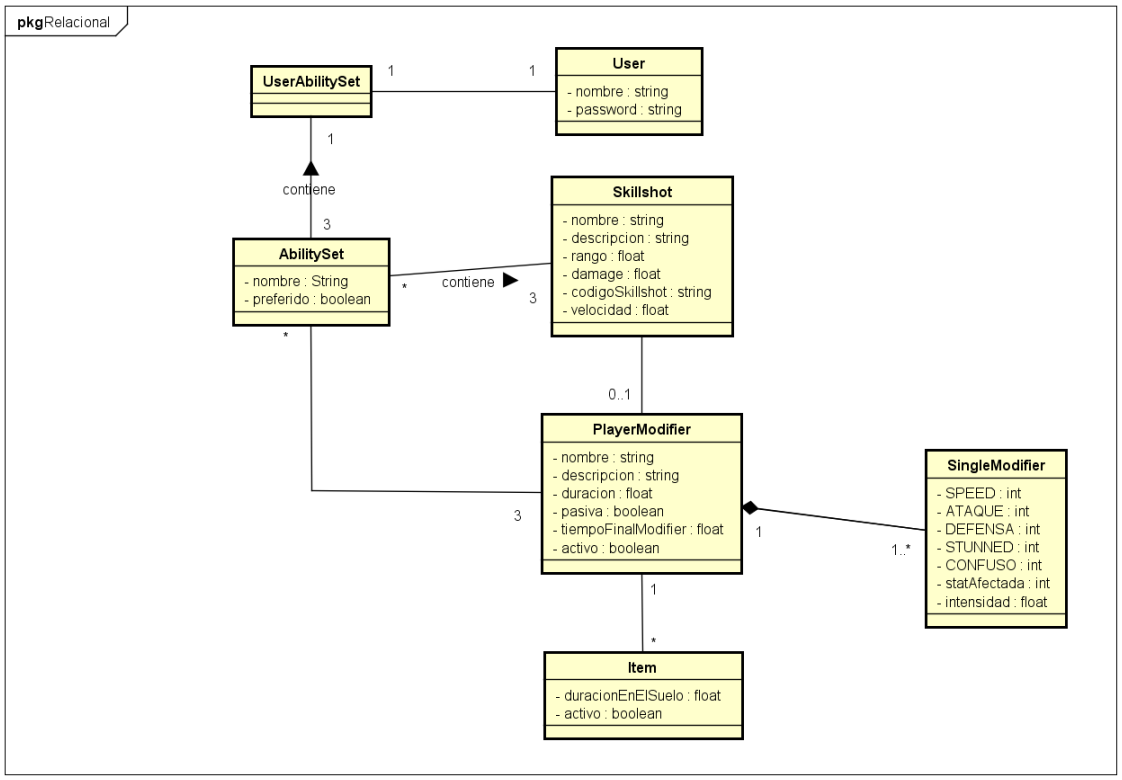


*Figura 6.8. Modelo relacional del sistema*

El modelo relacional representa solamente los elementos cuya información se considera necesaria de mantener de forma permanente en el sistema. Como podemos ver, toda la información de Usuarios y sus kits de habilidades es necesaria. También son necesarios el conjunto de todas las *Skillshot*, *PlayerModifier* (junto con sus *SingleModifier*) e *Items* existentes en el videojuego.

## 6.6. Diagramas de secuencia

Con el objetivo de facilitar el entendimiento y la implementación de los elementos más completos dentro de nuestro videojuego, vamos a realizar varios diagramas de secuencia pertenecientes al flujo habitual de cada uno de ellos.

## 6.6.1. Diagrama de secuencia de Player

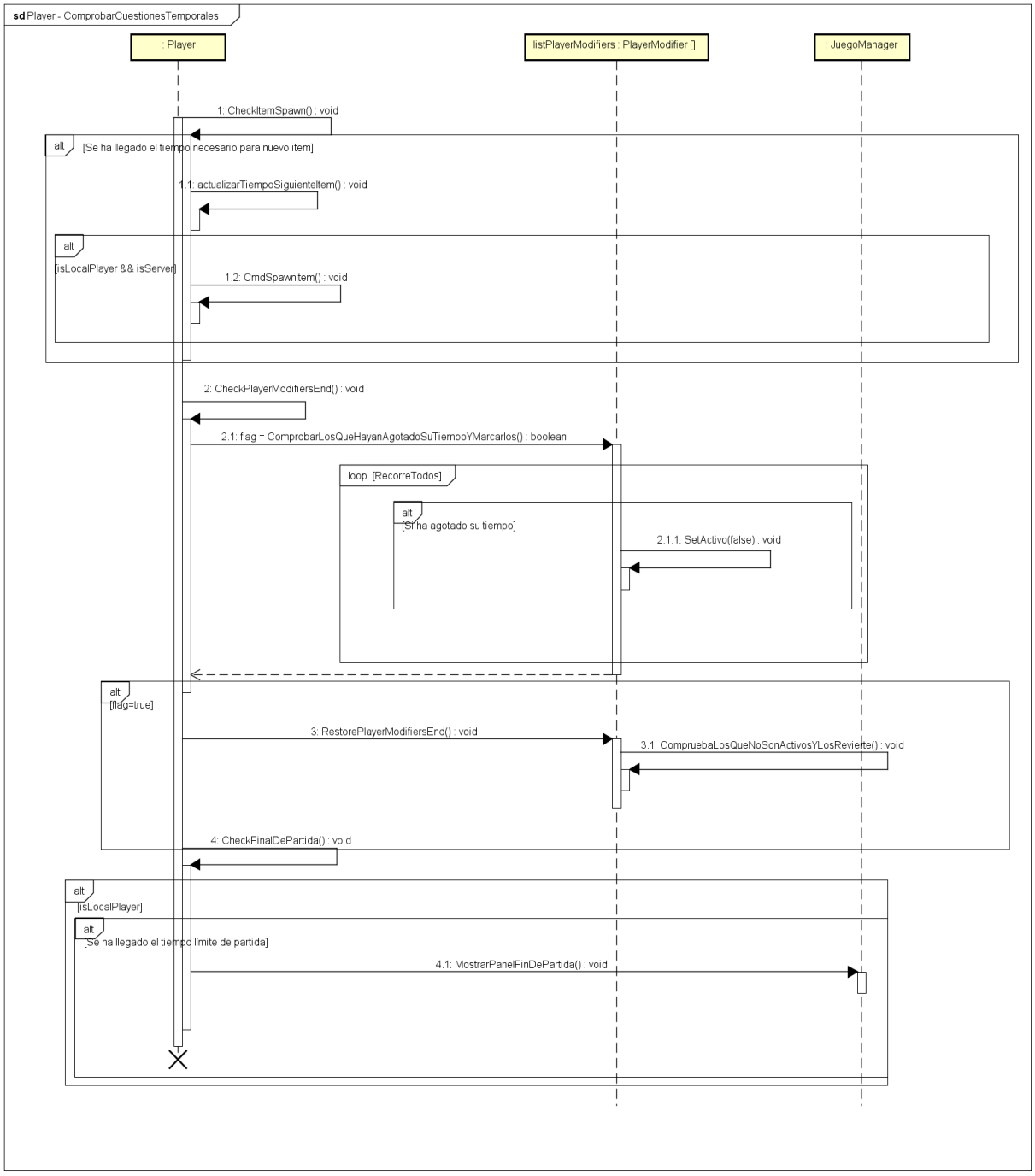


*Figura 6.9. Diagrama de Secuencia Player – Comprobacion de cuestiones temporales*

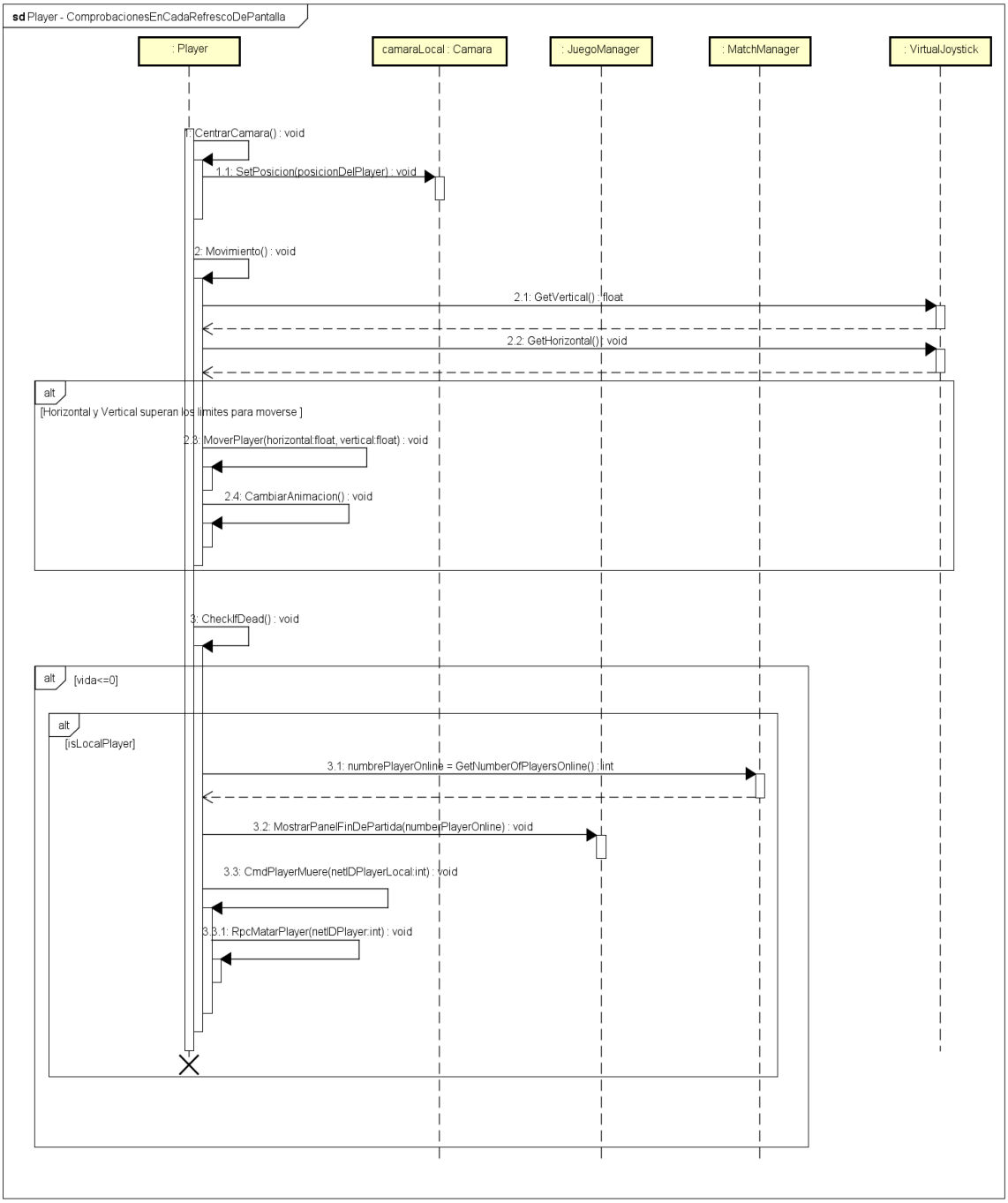


*Figura 6.10. Diagrama de Secuencia Player – Comprobaciones en FixedUpdate*

## 6.6.2. Diagrama de secuencia de SkillshotProjectile

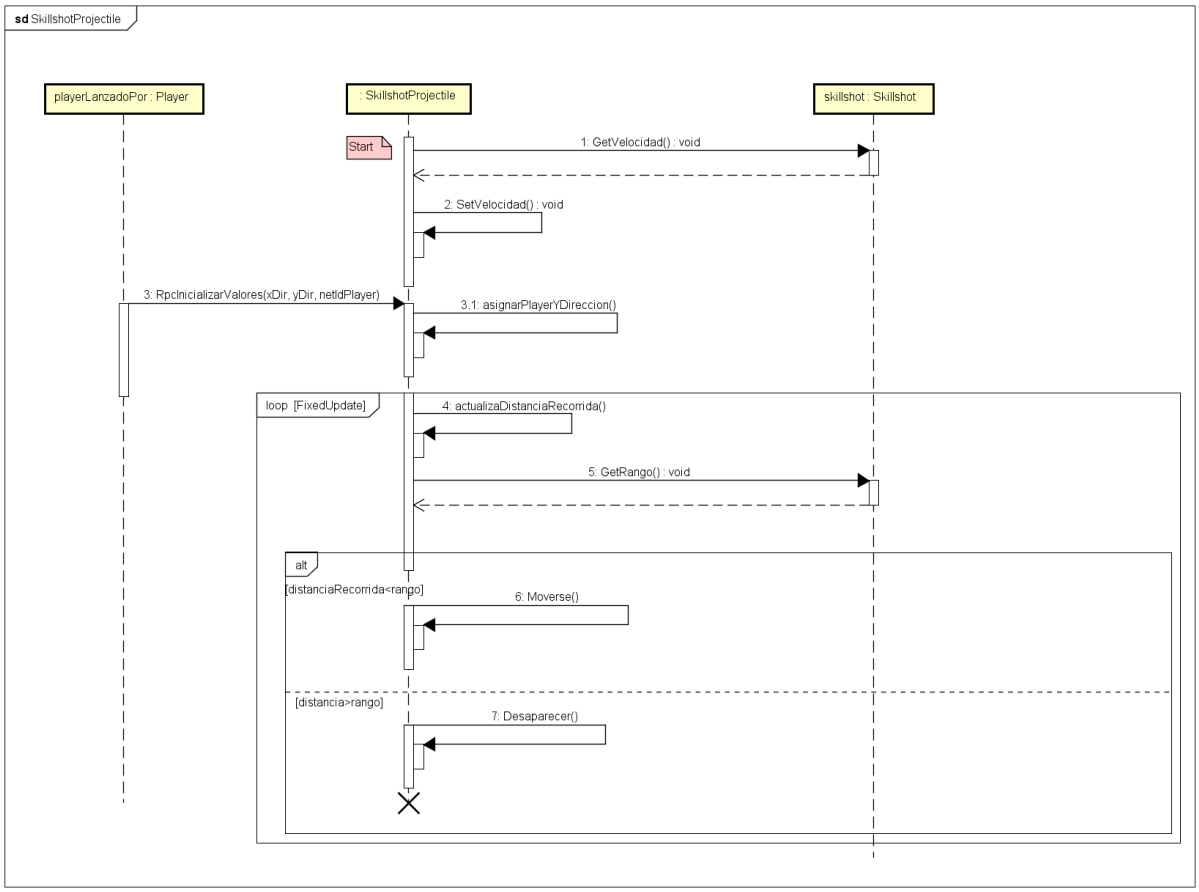


*Figura 6.11. Diagrama de secuencia de SkillshotProjectile*

# 7. Implementación

El desarrollo de videojuegos requiere de más tareas que la simple programación de código. Obviamente, es una parte muy importante y vital para que el videojuego funcione tal y como deseamos, pero para que el videojuego sea, entre otras cosas, atractivo a la vista, no podemos olvidarnos de las demás actividades necesarias. Entre ellas el manejo de animaciones, la creación manual de elementos de la escena desde el editor de Unity, la creación de los elementos visuales con herramientas como Piskel o la gestión de la base de datos. Esta última se hará a través de una clase auxiliar llamada SQLiteManager, que explicaremos más adelante.

## 7.1. Jugabilidad

El videojuego es un sistema interactivo un tanto especial, ya que está orientado a divertir y entretener al usuario, no a realizar tareas para un fin productivo necesariamente. Es difícil medir la calidad del software cuando el objetivo del videojuego es el de hacer sentir bien al jugador. Así que, se propone como posible medida para ello la jugabilidad. Ya hemos mencionado varias veces a lo largo de la memoria este concepto, pero no lo hemos desarrollado en detalle.

Como podemos ver en la siguiente figura, extraída de este artículo [47] [48] se agrupan varios parámetros, sensaciones o incluso sentimientos dentro de cinco grandes grupos, de los cuales mencionaremos los tres que más nos interesan.

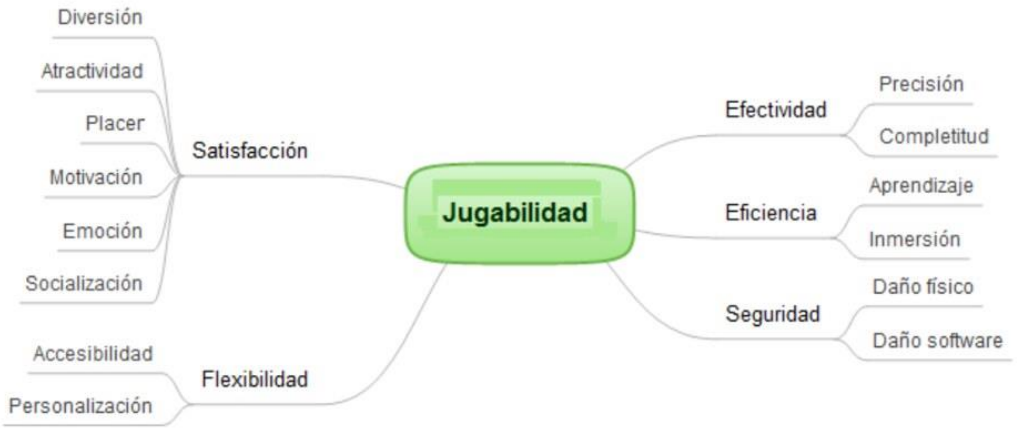


*Figura 7.1. Parámetros de la jugabilidad*

• **Eficiencia**:

Si hablamos de eficiencia en el campo de la usabilidad nos estamos refiriendo a realizar el mayor número de acciones correctamente por parte del usuario en el menor tiempo posible, por lo que el periodo de aprendizaje no es precisamente eficiente en ese sentido. Sin embargo, en el campo de la jugabilidad, el autor del artículo relaciona la eficiencia y efectividad con el tiempo y los recursos necesarios para lograr los objetivos propuestos en el videojuego, algo que se relaciona directamente con el aprendizaje.

Por tanto, un juego debe ser eficiente en cuanto al aprendizaje. Un buen diseño de videojuego no puede permitir que los usuarios pierdan horas sin aprender a jugar al juego, resulta frustrante y poco atractivo. Hay que buscar un juego sencillo de entender, con reglas marcadas, pero a su vez divertido.

• **Flexibilidad**:

Un juego que sea fácilmente accesible a la vez que permita una personalización dentro del juego con la que cada usuario pueda sentirse cómodo hacen que el juego sea flexible y aumente la jugabilidad.

Hemos querido que nuestro juego tenga un grado de personalización alto, por ello el usuario puede personalizar prácticamente todo en el juego. Desde el set de habilidades que quiere hasta el color y nombre que quiere que tenga su personaje en partida.

• **Satisfacción**:

Sentimientos como la diversión, el placer y la emoción son los que hacen al usuario disfrutar con el juego. La motivación viene dada por la satisfacción generada con el juego y las ganas de seguir jugando y aprendiendo de él, mejorando. Todo esto hace que el usuario quede satisfecho con el juego.

El componente competitivo de nuestro juego es el que le da la emoción y motivación para seguir jugando, unido a la socialización (al ser en red), que potencia las dos anteriores.

Hemos querido cumplir con estos atributos en el diseño del videojuego para que acabe teniendo una jugabilidad buena. En el siguiente apartado hablaremos sobre los prototipos que se han ideado para que, mientras que cumple estos tres puntos deseables mencionados, el juego en pantalla apoye y mejore esto, con una usabilidad buena.

## 7.2. Prototipos

A la hora de diseñar la disposición de los elementos en la pantalla, hemos creado varios prototipos de cómo gustaría que quedara el producto final.

En cuanto a la apariencia deseable que debería tener la personalización de los AbilitySet, hemos realizado dos prototipos, muy similares. Uno para la parte de personalizar las *Skillshot* y otro para la de las pasivas. Se muestran en las dos siguientes figuras, respectivamente.

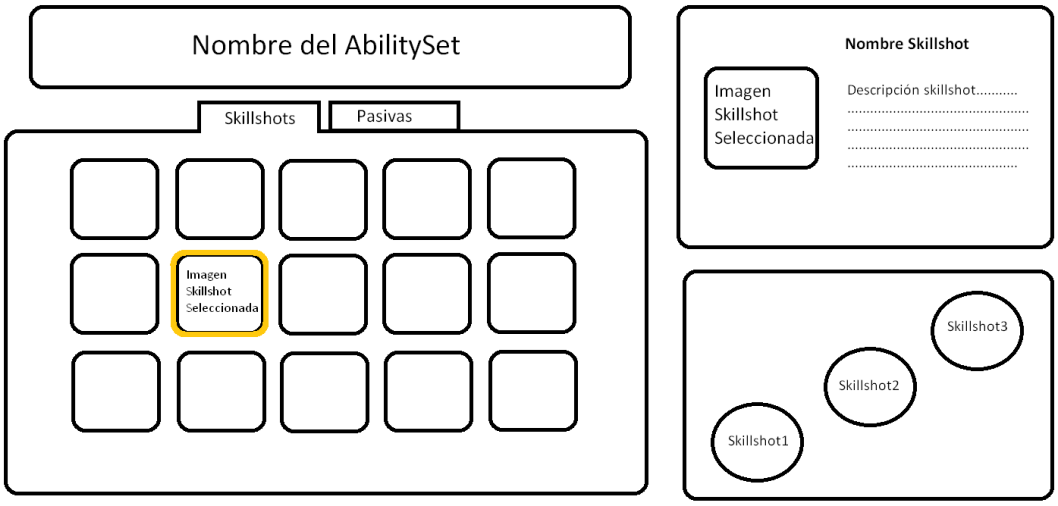


*Figura 7.2. Prototipo de pantalla de personalización de las skillshots*

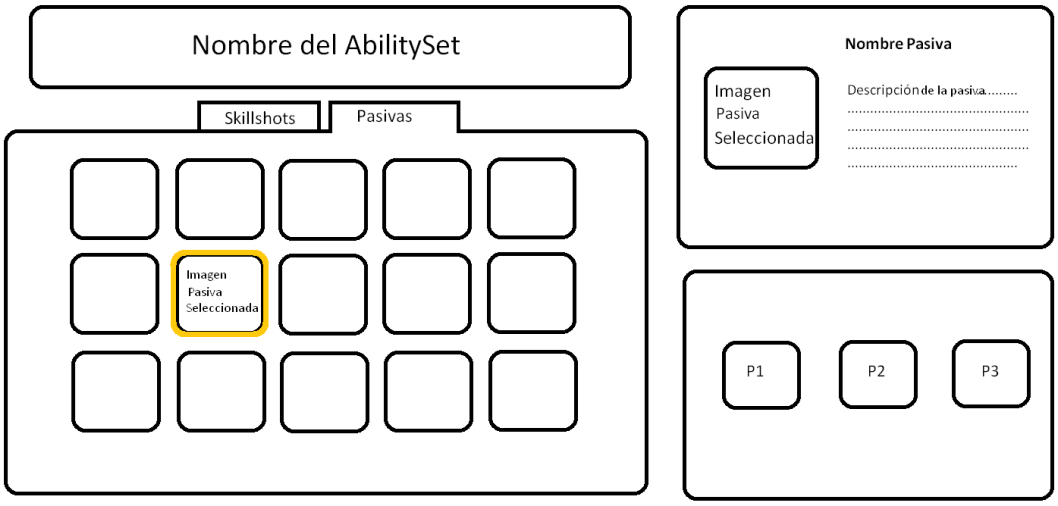


*Figura 7.3. Prototipo de pantalla de personalización de las pasivas*

Nos ha parecido una buena decisión en cuanto a la distinta forma de colocar las pasivas y *Skillshot* elegidas, el hacerlo de la manera exacta en que se hace. La razón es que esa misma disposición es la que van a tener en el juego, cuyo prototipo veremos a continuación, y que al usuario le ayudará a relacionar de una manera más clara qué parte de su juego es el que está editando. Va a saber que cuando esté editando las *Skillshot* (los circulares) son para las habilidades lanzables, y cuando edite las cuadradas, serán para las habilidades pasivas.

En cuanto al prototipo de la pantalla de juego, hemos sido conscientes del limitado tamaño que tenemos al ser un juego en un dispositivo móvil. Por esa razón, hemos decidido investigar y tomar como ejemplo otros videojuegos con temática similar para ver cómo han resuelto el problema.

Hemos tenido que encontrar un juego en el que importen el tamaño de los personajes, para que se puedan ver correctamente en pantalla y a su vez que haya varios joysticks que nos permitan lanzar las habilidades. Todo considerando que esto último no nos quite visibilidad del personaje. Este ha sido el motivo principal por el cual se ha decidido que la cámara siga en todo momento al personaje, de manera que quede situado siempre en el centro. El juego principal que nos ha servido de referencia ha sido el Brawl Stars.


*Figura 7.4. Partida de Brawl Stars*

Como podemos ver, utiliza un *joystick* a la izquierda de la pantalla, que sirve para el movimiento, y otros dos más pequeños en la parte derecha de ésta. También podemos ver el tamaño de los personajes, que es el idóneo para la jugabilidad.

En cuanto a esos aspectos, Brawl Stars ha sido de gran ayuda. En cambio, hay otros elementos, contenidos también en el HUD, que el Brawl Stars no nos proporciona. Desde un principio queríamos que el jugador pudiera ver sus estadísticas base, su barra de vida y los PlayerModifier que le están afectando actualmente, aparte de sus Pasivas. Hay un juego que cumple casi al 100% estos requisitos, la única pega es que no es para dispositivo móvil, pero aún así creemos que muestra esa información de una forma tan buena y poco intrusiva en la pantalla de juego que lo hemos querido tomar como referencia. Es el caso del League Of Legends.

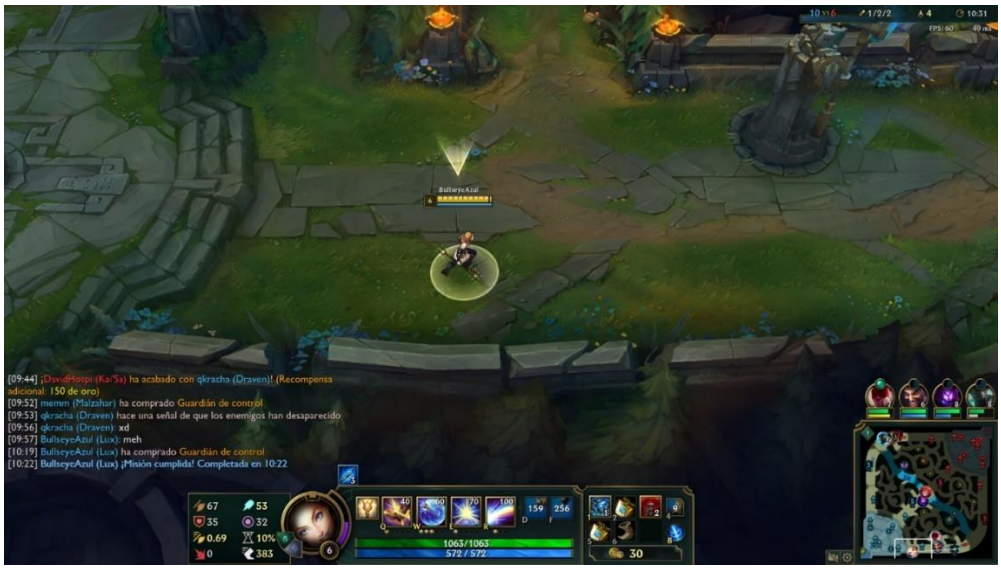

*Figura 7.5. Partida con pantalla completa de League Of Legends*


*Figura 7.6. HUD visto con detalle de League Of Legends*

Como podemos ver en la imagen, este juego nos proporciona los 3 elementos que queríamos incluir en nuestro HUD. Tomaremos en cuenta esta manera de hacerlo para incluirlo a nuestra pantalla de juego, además de añadir las pasivas, cosa que no aparece en el League of Legends, pero creemos que bajo la barra de vida puede ser una buena posición. A continuación, mostramos el prototipo ideado para la pantalla de juego.


*Figura 7.7. Prototipo detallado de la pantalla de juego*

También, como podemos observar en ambos juegos, se ha decidido añadir una pequeña barra de vida y un nombre justo encima del jugador, para dar una visión más rápida de la salud de nuestro personaje, tanto para nosotros como jugadores como para el jugador contrario.

A continuación, describiremos los elementos de la fase de implementación más destacables.

## 7.3. Interfaz de usuario fuera de partida

En este punto vamos a hablar de la implementación y diseño de la interfaz de usuario fuera de la partida, es decir, de los diversos menús y escenas para personalizar las habilidades y acceder a la partida. La parte visual de estos menús se ha realizado con los elementos UI (User Interface) que nos brinda Unity, como son Canvas (contendor de elementos de UI) e Image (de base son imágenes, pero pueden incorporar componentes para darles más utilidad).

Cabe destacar que tanto dentro como fuera de la partida, en cada escena, un GameObject con un sencillo script (en la mayoría de los casos) para gestionar los elementos de UI. Por ejemplo, para volver hacia atrás de una escena a otra, para salir de la partida al Lobby, etc. Estos elementos suelen llamarse como la escena a la que pertenecer con el añadido final de Manager.

Nombraremos solamente aquellos elementos que hayan supuesto un costo adicional de implementación, ya que una gran parte de este desarrollo es el trabajo con el editor de Unity y consiste en añadir elementos a la escena, colocarlos y asignarles funciones al recibir eventos.

### 7.3.1. Personalización AbilitySet

Hemos considerado digna de mención esta escena, que, aunque lleva gran parte de edición mediante Unity, también incorpora las funciones por las que el usuario puede editar directamente su AbilitySet. Creemos que esto ayudará a entender la apuesta por la facilidad de aprendizaje a la hora de personalizar elementos dentro del juego.

A continuación, mostramos la jerarquía de la escena, esta es la más compleja de todas debido a su elevado número de elementos, pero necesario para explorar y editar todas las posibilidades que debería ofrecernos personalizar un AbilitySet. En esta primera imagen, veremos que dentro de Canvas_AbilitySet tenemos otros 4 Canvas.

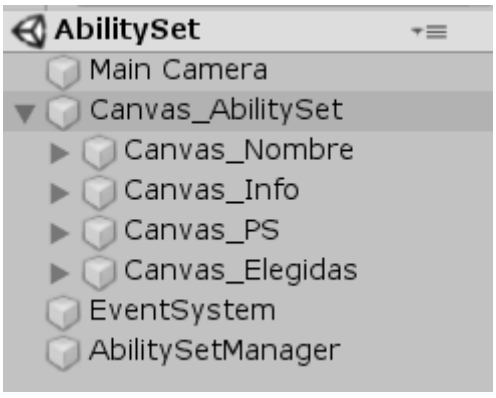


*Figura 7.8. Jerarquía escena AbilitySet*

A continuación, desarrollaremos individualmente el contenido de cada Canvas.

• Canvas_Nombre:

Este Canvas contiene tres elementos principales. De izquierda a derecha, el botón de volver a “Personalizar”, el nombre del AbilitySet y el botón para guardar de manera permanente los cambios realizados al AbilitySet.


*Figura 7.9. Elementos de la escena Canvas_Nombre*

En cuanto al botón de volver, su implementación es similar a la del apartado anterior, pero en vez de regresar al menú principal, regresa al menú inmediatamente anterior, el de Personalizar.

Respecto al nombre, al pulsar en él se habilitará un cuadro de texto para introducir el nuevo nombre que queramos ponerle al AbilitySet.

El botón de guardar llama a la función del manejador “GuardarCambios” la cual llama a la función en el SQLiteManager para que guarde toda la información.

• Canvas_PS:

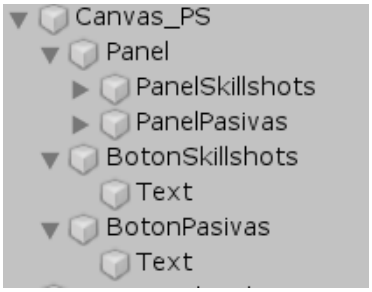


*Figura 7.10. Jerarquía Canvas_PS*

Dentro de este Canvas tenemos 3 principales elementos, los dos primeros y más sencillos son BotonSkillshots y BotonPasivas, después explicaremos el Panel principal.


*Figura 7.11. BotonSkillshots y BotonPasivas*

La funcionalidad de estos botones se resume en cambiar la apariencia de la escena en general para *Skillshots* o para Pasivas, según al botón que pulses, desactivará y activará los paneles correspondientes.

Con respecto al Panel principal, vemos que contiene, a su vez, dos paneles. El PanelSkillshots contiene imágenes de todos los *Skillshots* de los que el usuario dispone para seleccionar. El PanelPasivas contiene lo mismo, pero con las Pasivas.


*Figura 7.12. Interfaz para personalizar las skillshots*


*Figura 7.13. Interfaz para personalizar las pasivas*

Al pulsar cualquiera de las imágenes (que tiene funcionalidad de botón), la información de ese elemento se cargará en el Canvas_Info, que explicaremos a continuación. Además, ese elemento queda “marcado” o “elegido”, esto es importante para lo que explicaremos en el Canvas_Elegidas.

• Canvas_Info:

Este Canvas es bastante sencillo, contiene un panel en el cual viene cierta información sobre la Pasiva o *Skillshot* que hayamos "elegido" en el Canvas_PS. Entre la información tenemos el nombre, la descripción y la imagen. La imagen siguiente es un ejemplo en vivo del resultado final.


*Figura 7.14. Canvas_Info*

• Canvas_Elegidas:


*Figura 7.15. Canvas_Elegidas*

En este Canvas tenemos algo similar a lo que teníamos en Canvas_PS, tenemos un Panel para los *Skillshot* y otro para las Pasivas. Cuál de los dos se muestra, se maneja con las pestañas que describimos en Canvas_PS. La funcionalidad de esta parte está centrada en qué *Skillshots* y Pasivas son las que tenemos actualmente seleccionadas en el AbilitySet. En las 3 ranuras de cualquiera de los elementos, si pulsamos en cualquiera de ellas teniendo seleccionado o "marcado" un elemento del Canvas_PS, se pondrá su imagen inmediatamente en la ranura seleccionada.

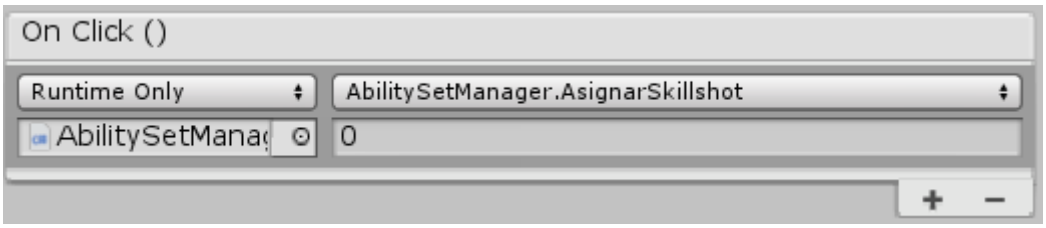


*Figura 7.16. Asignación del Onclick de Skillshot*

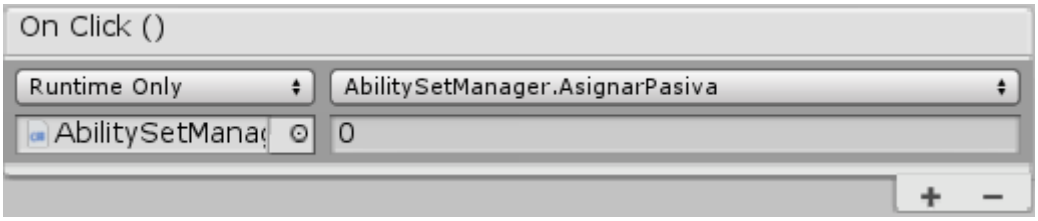


*Figura 7.17. Asiganación del Onclick de Pasiva*

Como vemos en la imagen, cada ranura tiene esta función, en la que coloca el *Skillshot* o Pasiva "marcado" en la ranura correspondiente, cambiando así, a través de esa función del manejador, el objeto AbilitySet que realmente estamos editando.

## 7.4. Interfaz de usuario dentro de la partida

Llegados a este punto conviene explicar un elemento muy útil de Unity, el Prefab. El sistema Prefab de Unity nos permite crear, configurar y almacenar un GameObject completo (incluso con jerarquía compleja en su interior) como un elemento reutilizable. De este modo, cada vez que se desee que aparezca un elemento, del cual podemos querer varias copias, ya tenemos este objeto "prefabricado" listo para instanciar, sin necesidad de volver a crear este elemento y configurarlo desde cero.

En este apartado vamos a abordar la implementación de lo que es la partida en sí misma, el juego. Empezaremos por el personaje, pasaremos por el HUD y acabaremos hablando de todos los elementos que aparecen en el juego.

### 7.4.1. Personaje

Es el elemento más importante de la partida y en torno al cuál va a girar. El personaje está representado por la clase Player, y conformado como Prefab con el mismo nombre. Dentro de su jerarquía simplemente tenemos una barra de vida que va por encima del personaje y que en el Script se va a usar para poder

ver de forma gráfica la vida restante. Vamos a echar un vistazo a los componentes.

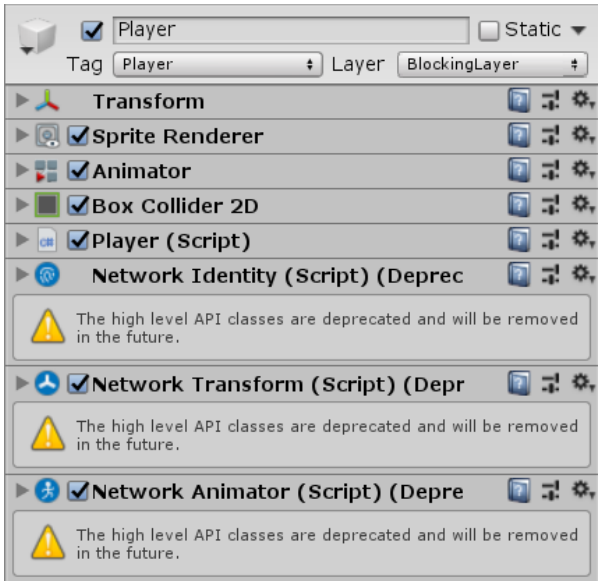


*Figura 7.18.Componentes de Player*

• **Transform**, como explicábamos en el apartado de Diseño, es un componente que se encarga de manejar la posición, rotación y escala del Player.

• **Sprite Renderer** es el encargado de renderizar y mostrar al personaje en la pantalla.

• **Animator** es el encargado, junto con el AnimatorController, realizar las animaciones para simular que el personaje se orienta correctamente dependiendo a la dirección a la que vaya, así como que muestre la animación correcta cuando ataque.

• **Box Collider 2D** describe el área de colisión del personaje, se ajusta manualmente para que resulte lo más parecido al área que ocupa el Sprite del personaje.

• **Network Identity** es necesario para la funcionalidad en red, es la forma de identificarse y diferenciarse de otros objetos dentro de la red.

• **Network Transform** es necesario para que el Transform del Player funcione correctamente en la red.

• **Network Animator** es necesario para que el Animator del Player funcione correctamente en la red y sea visible para los demás usuarios.

Y, finalmente, el Script **Player**, que describe todas las funciones que va a realizar el personaje, así como interacciones con los demás elementos de la partida.

Primero vamos a mencionar los atributos del personaje, los cuales van a ser de vital importancia para el desarrollo de la partida.

• **Vida** y **Vida Máxima** serán los atributos de vida restante del personaje y vida con la que comienza la partida, respectivamente.

• **Ataque** y **Defensa** serán los atributos de ataque y defensa del personaje, estos atributos son importantes a la hora de calcular el daño recibido por habilidades de otros jugadores.

• **Velocidad** es un atributo heredado de MovingObject, representa la velocidad con la que el personaje va a moverse por el campo.

• **Confuso** y **Stunned** son dos atributos booleanos que representarán si el personaje sufre alguna de estos dos "dolencias". Confuso afecta al personaje de tal manera que tanto su movimiento como su lanzamiento de habilidades se realiza en la dirección totalmente opuesta. Stunned, en español, aturdido, incapacitará al personaje para moverse y para lanzar habilidades durante todo el tiempo que este atributo se encuentre activo.

Los mencionados anteriormente son atributos "de partida" del jugador, son los que pueden ser modificados por los PlayerModifier, pero tenemos más atributos que son importantes para el correcto funcionamiento de la partida.

• **AbilitySet** es el AbilitySet con el que se ha decidido jugar esta partida, las *Skillshots* y Pasivas que este contenga serán las que el Player puede lanzar.

• **HUD** será el HUD del que dispondrá nuestro personaje, mostrará toda la información útil, además de proporcionar los distintos *joysticks* para que el usuario pueda mover al personaje y lanzar habilidades.

• **ListPlayerModifiers** será la lista cambiante de PlayerModifiers que están afectando en ese determinado momento a nuestro Player. Durante toda la partida tendremos al menos 3, nuestras "pasivas".

En cuanto a las funciones más importantes dentro de Player podemos encontrar las siguientes.

• **InicializarEstadisticas** se encarga de inicializar las estadísticas con unos valores fijos para todos los jugadores, que luego pueden ser y serán afectados por los PlayerModifier.

• **InicializarYAsignarHUD** se encarga de inicializar y asignar el HUD y todo lo necesario para que éste arranque correctamente cargando todos los datos necesarios. Esta función solamente la va a llamar cada jugador local en su

dispositivo, con ayuda de la comprobación *isLocalPlayer*, hablaremos de ello más adelante.

• **AffectPlayerModifier** recibe un PlayerModifier y se encarga de aplicarlo al Player, alterando sus atributos con ayuda de los SimpleModifier de los que el PM se compone. También calcula el tiempo en el que éste ha sido aplicado y autocalcula, con ayuda de la duración del PM, el instante del tiempo en el que el PM debería dejar de aplicarse, y, por tanto, revertirse.

• **CheckPlayerModifiersEnd** se ejecuta de forma periódica y se encarga de comprobar si algún PM que afecte al Player ha llegado ya a su tiempo de dejar de aplicarse, si es así, marca el índice del PM para que se revierta en la siguiente función que explicaremos. Si ha encontrado alguno, esta función devolverá *true*. La razón para hacerlo de esta manera es la de mejorar la eficiencia del videojuego.

• **RestoreStatsPlayerModifiersEnded** se ejecuta de forma periódica pero solamente cuando la función anterior (CheckPlayerModifiersEnd) ha devuelto true, que significa que la función que estamos explicando tiene algo que hacer, si devuelve false, simplemente sería perder tiempo de computación ya que no tendría nada que hacer. De esta manera nos ahorramos que esta función se ejecute 50 veces por segundo, haciendo que se ejecute solamente cuando un PlayerModifier deja de afectar al personaje, situación que, normalmente, pueden pasar varios segundos sin ocurrir.

• **UpdateHealthBar** se encarga de actualizar la barra de vida que tiene el Player justo encima de él en la escena, haciendo un calculo con la vida máxima y el tamaño de la barra de vida.

• **DoDamage** función que se usará para restarle vida al personaje, para aplicar el daño que pueda venir de diversas fuentes, las más normales, *Skillshots* de otros jugadores.

• **AffectSkillshot** recibe una *Skillshot* (la que le ha impactado al Player) y el lanzador de ésta. Calcula el daño y los PlayerModifiers que puedan afectarle y lo aplica.

• **CentrarCamara** es una función que se encarga de mantener la cámara del juego siguiendo en todo momento al jugador.

• **CheckIfDead** se encarga de comprobar periódicamente si la vida del Player ha bajado de 0 y si es así mostrarle un mensaje al jugador de que la partida ha terminado para él.

• **CheckMorePlayersLeft** se encarga de comprobar si hay más jugadores en el campo de batalla, y si no los hay, declarar al jugador local como vencedor.

• **CheckFinalDePartida** se encarga de comprobar si se ha llegado al tiempo límite de partida, y, si es así, notificar al jugador y terminarla.

Estos tres últimos métodos, como hemos visto en el diagrama de actividad de Player, son las condiciones de finalización de la partida. Si el personaje local muere, se llega al final de tiempo límite de la partida, o incluso, si el nuestro personaje local es el último en pie, la partida habrá finalizado y por tanto se mostrará un mensaje al usuario indicándole su posición en el podio. Este sistema de detectar qué posición ha obtenido nuestro personaje, se realiza con una clase llamada **MatchManager**, la cual contiene un array de *networkIdentities* cada una perteneciente a cada jugador online que se encuentra jugando la partida. Cuando un jugador muere, se llamará a una función que más tarde explicaremos, para notificar al resto de jugadores que ha dejado de jugar, y que eliminen su *networkIdentity* de su array.

Hasta aquí son métodos a nivel local de la partida, en el apartado de red comentaremos en profundidad los métodos que tiene Player con este carácter.

Esas son las funciones más importantes en cuanto a funcionalidad característica de Player, pero aún hay más funcionalidad del personaje que explicaremos en apartados siguientes, como el movimiento o las animaciones.

### 7.4.2. HUD

El HUD (Head-Up Display), conocido en español como barra de estado, es la información mostrada por pantalla de la que dispone el usuario durante el transcurso de la partida.

Es una clase implementada desde cero y conformada por el conjunto de los elementos necesarios para mostrar toda la información necesaria en partida al usuario.

En nuestro caso el HUD, está compuesto por los siguientes:

• **Estadísticas del jugador** estadísticas visibles del jugador como son ataque defensa y velocidad.

• **Barra de vida** una barra de vida más grande que la que acompaña a cada Player, la cual se va actualizando con la vida actual del jugador actual.

• **Pasivas,** que son 3 PlayerModifiers que durarán toda la partida, y serán visibles en todo momento debajo de la barra de vida.

• **PlayerModifiers,** una lista de todos los PM que afectan al usuario en ese instante, menos las 3 pasivas, que tienen su lugar reservado. Estos se sitúan justo encima de la barra de vida y desaparecen cuando se les agota el efecto. Útiles para que el usuario sepa por qué PM está afecta en ese momento.

• **VirtualJoystick de movimiento** es un *joystick* situado a la parte izquierda de la pantalla que permite al usuario mover a su personaje.

• **VirtualJoystick de habilidades** son 3, y son *joysticks* situados a la parte derecha de la pantalla. Permiten al usuario lanzar las *Skillshots* que el jugador haya asignado al personaje.


*Figura 7.19. Vista general del HUD en partida*

Ahora entraremos a explicar la jerarquía del HUD y su correspondencia con los elementos que hemos mencionado anteriormente.


*Figura 7.20. Jerarquía del HUD*

• **PlantillaStats** simplemente es una imagen con los símbolos de ataque, defensa y velocidad y los recuadros de debajo, dejando libres los cuadros de la derecha para un TextStats (clase para mostrar los atributos de los Players), que

simplemente es un GameObject con un componente de tipo Text. Se corresponde con **estadísticas del jugador**.


*Figura 7.21. Plantilla de las estadísticas*

• **Barra de vida** contiene una imagen verde y una imagen como el rectángulo que podemos ver, pero vacío. Al actualizarse con la vida actual, la barra de la imagen verde se va haciendo más pequeña, dejando solo a la vista la parte vacía. El efecto que produce es el siguiente.

*Figura 7.22. Barra de vida*

• **PlantillaPasivas** contiene la imagen con recuadros del fondo y 3 imágenes que se ocuparán con las pasivas elegidas por el jugador.

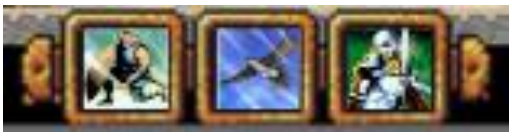

*Figura 7.23. Plantilla que aloja las pasivas*

• **PlantillaPlayerModifier** contiene varias imágenes que estarán invisibles hasta que, las posiciones del ListPlayerModifiers del Player tenga más de 3 (hasta ahí está reservado para las Pasivas) PlayerModifiers. En ese momento las imágenes de los nuevos PM que vayan afectando al jugador se irán mostrando, ocupando las imágenes hasta entonces invisibles.

• **JoystickBackgroundImage** contiene un script de tipo *VirtualJoystick*, el encargado de calcular el movimiento que va a realizar el personaje según donde se coloque el puntero (otra imagen, pero más pequeña).

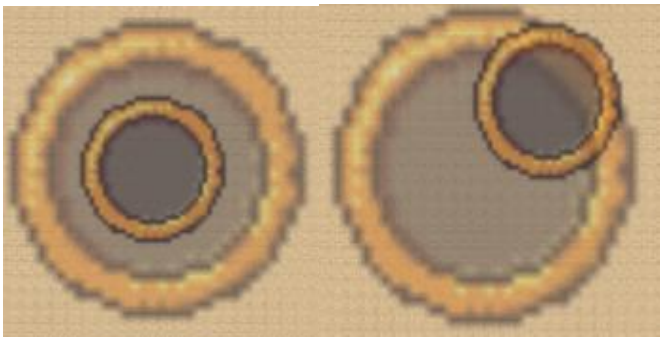

*Figura 7.24. Joystick de movimiento en acción*

• **Canvas_Skillshots** es un Canvas que contiene los 3 *JoystickSkillshotBackgroundImage* pertenecientes a cada uno de los 3 *Skillshots* que va a poder utilizar el usuario durante la partida. En cuanto a composición en jerarquía además de lo mencionado en el *joystick* de movimiento disponen de una imagen para mostrar a qué *Skillshot* pertenece cada *joystick*.

*Figura 7.25. Joystick de habilidades en acción*

Una vez vistos todos los componentes y su importancia de cara a la información que tiene que recibir el usuario, pasaremos a explicar las funciones más importantes dentro del script HUD.

Una de las cosas más importantes dentro del script es la “recogida” correcta de todos los elementos creados desde el editor en el script, todo con ayuda del comando Find, explicado en la parte de diseño. Una vez todos los elementos relevantes de la escena tienen su objeto en el script, simplemente las funciones son **ActualizaImagenesPlayerModifiers**, **ActualizarStats**,

**ActualizarImagenesPasivas**, **ActualizarVida** y **ActualizarSkillshots**. Como sus nombres indican, las funciones se encargan de ir rellenando correctamente los elementos descritos en el HUD. Como mención importante a parte de estas funciones tenemos **UpdateHUD** y **SoftUpdateHUD**. Básicamente lo que hacen estas funciones es ejecutar las primeras funciones nombradas, con la diferencia de que UpdateHUD ejecuta todas y SoftUpdateHUD solo ejecuta las que suelen cambiar más a menudo. El motivo de esto vuelve a ser la optimización. Hay alguna función que requiere acceso a archivos, como es el caso de ActualizaSkillshots que muy rara vez va a ser llamado, excepto al empezar la partida. En cambio, algunas otras como ActualizarStats y ActualizarVida sí van a cambiar muy a menudo, de ahí el separarlo en dos funciones distintas.

### 7.4.3. SkillshotProjectile

Es otro de los elementos más importante de la partida, ya que es la manera principal que tienen para interactuar unos personajes con otros y ganar la partida. El lanzamiento de la habilidad está representado por la clase *SkillshotProjectile*, y conformado como *Prefab* con el mismo nombre. No tiene jerarquía, está conformado por solo un *GameObject* y en cuanto a sus componentes tiene exactamente los mismos y por los mismos motivos que Player, con la salvedad de que el script en este caso es el de *SkillshotProjectile*.

La vida útil del *SkillshotProjectile* se entiende desde que sale de la ubicación del Player que la lanza hasta que, o bien impacta en un contrario y bien agota su distancia de recorrido (conocido como rango), que desaparece.

La función más importante de esta clase se mencionará en el apartado de red, la siguiente función es importante a nivel gráfico del videojuego.

• **GetRotacionByDirection** nos devuelve el ángulo de rotación que tiene que tener ese *Skillshot* dependiendo de su dirección, de manera que favorezca a una coherencia visual y una experiencia de juego más real.

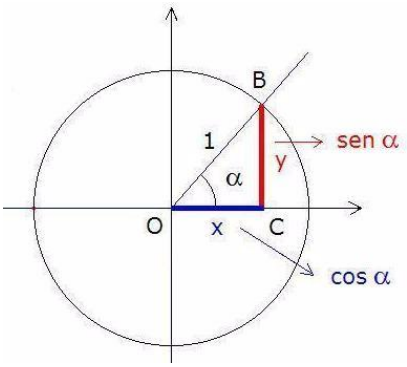


*Figura 7.26. Representación del seno y coseno Fuente: [49]*

Suponiendo el eje x (dirección horizontal) como el cos(α) y el eje y (dirección vertical) como sen(α), para hallar el ángulo α que necesitamos para calcular la rotación, hallaríamos la tangente diviendo el sen/cos y después aplicándole la función `Mathf.Atan(tangente)` hallaríamos el ángulo deseado.

Para que la rotación sea siempre la misma vamos a procurar todas las animaciones de las *SkillshotProjectile* orientadas hacia la derecha, para que la función y cálculo de la rotación sea válido para todas ellas, como vemos en estos ejemplos:

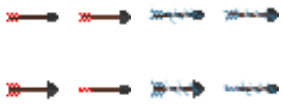

*Figura 7.27. Orientación inicial que deben tener todas las SkillshotProjectile*

En cuanto a cómo se instancian estos elementos en el juego en red, será el *VirtualJoystickSkillshot* el encargado de empezar las llamadas pertinentes para que sea lanzado, cuando detecte en su función `OnPointerUp` que su cursor o puntero ha sido movido hacia algún lado del *joystick* y posteriormente se ha soltado, gesto que el usuario tiene que hacer si quiere lanzar la habilidad.

### 7.4.4. Items

Los items, u objetos, son elementos, que, aunque no sean vitales para el desarrollo normal de la partida, sí le añaden un poco más de interactividad y amplían las posibilidades dentro de una partida. Como los mencionados anteriormente también son Prefab, ya que a lo largo de la partida habrá varios de estos elementos apareciéndose de forma aleatoria en el campo de batalla. Este comportamiento es el que se va a tratar de realizar gracias a las funciones que describiremos a continuación. En cuanto a la jerarquía no posee nada más. Y en componentes, son exactamente los mismos que *Skillshot* y Player.

Como mencionamos en el apartado de diseño, ítem es la clase con los métodos pero el “envoltorio” para que pueda funcionar en la partida es ItemInstance.

Las funciones relevantes son:

• **GetItemPlayerModifier** que devuelve el PlayerModifier asignado al *Item* en cuestión.

• **CatchItemSimple** que hace desaparecer al *Item*, una vez hemos recogido anteriormente su PlayerModifier.

Y hasta aquí los *Prefabs* importantes para la partida y la implementación de sus scripts. A continuación, hablaremos de otras características muy importantes

también para el videojuego, como animaciones, comandos especiales de red y entrar a explicar cómo funciona el movimiento con los *VirtualJoysticks*.

### 7.4.5. Animaciones

Las animaciones que se tienen en la escena se gestionan principalmente con el componente *Animator*. Este componente hace uso de un *AnimatorController*, el cual describe un comportamiento, en forma de máquinas de estados. Según determinadas variables que nosotros indicaremos, el controlador hará que el GameObject pase por un estado o por otro, reproduciendo en el estado en el que esté la animación correspondiente.

Para crear las animaciones en 2D basta con conseguir un pequeño grupo de imágenes, que unidas secuencialmente como si de un GIF se tratase, simulen el movimiento del *GameObject*, por ejemplo, aquí podemos ver las de una *SkillshotProjectile*.

*Figura 7.28. Ejemplo animación de SkillshotProjectile*

Aunque muy parecidas, las imágenes son un poco distintas y gracias a eso, cuando se reproducen de forma secuencial, da la impresión de que la bola de fuego está en continuo cambio mientras viaja por el campo de batalla. Algunas de estas imágenes han sido adquiridas mediante licencias Creative Commons, otras han sido dibujadas por el equipo de desarrollo.

A continuación, mostraremos el AnimatorController del Player para explicar cómo funciona.

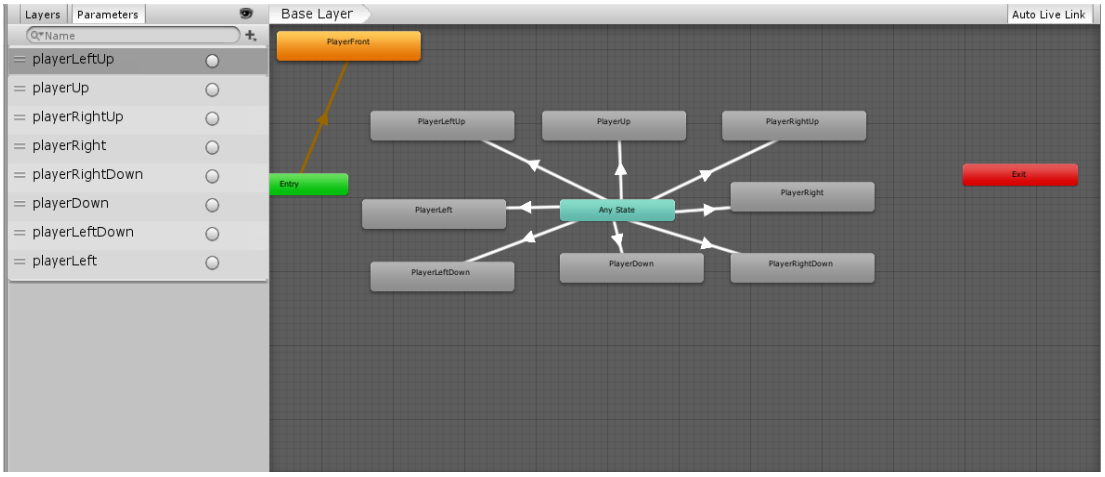


*Figura 7.29. PlayerController, el AnimatorController de Player*

Como podemos ver, el AnimatorController consiste en una máquina de estados, la cual empieza en un estado base, que será cuando el Player esté ocioso, sin movimiento ni ataque.

Si nos fijamos en la parte izquierda de la imagen, hay unas algunas variables. Éstas en concreto se denominan Triggers. Desde los scripts podemos desencadenar estos triggers, eso lo escucha el Controller y activa ese Trigger.

Como vemos desde Any State, podemos ir a cualquiera de los otros estados. Lo que necesita para ir de uno a otro se denomina transición. Veamos a fondo qué es una transición.

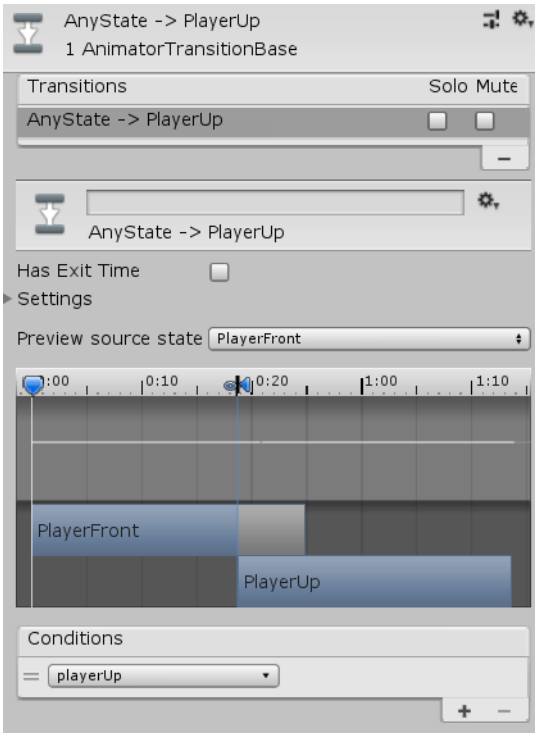


*Figura 7.30. Ejemplo de transición de Player*

Una transición representa tanto las condiciones como la forma en la que pasar de un estado a otro. Como podemos ver, ésta va de AnyState, es decir, de cualquier otro estado, hasta PlayerUp. En la configuración tenemos varios parámetros para adecuar la transición a nuestro gusto. Por ejemplo, si queremos que la transición dure más o menos, que sea inmediata, que tenga algún tipo de retardo, etc.

Por último, vemos que tiene “Condiciones” para que esta transición se ejecute. La única y simple condición de esta transición es que el trigger “playerUp” sea lanzado.

Como decíamos antes, los triggers se pueden desencadenar desde los scripts, en este caso concreto, desde el script de Player, en el momento previo a realizar un movimiento, calculando la dirección que el Player va a tomar, ejecutamos un

trigger u otro, para darle al personaje la animación coherente con la dirección que éste va a tomar.

### 7.4.6. Movimiento

En este punto volvemos a hablar de programación, concretamente de cómo funciona el movimiento tanto del personaje como de los lanzamientos de habilidades.

Lo primero que debemos saber es que ambas clases heredan de una abstracta llamada MovingObject (Objeto Movible). Éste describe el comportamiento que debe tener un objeto que quiera moverse por el terreno, con qué velocidad y qué hacer cuando se topa con otros elementos.

En cuanto a las funciones implementadas, las más notables son **AttemptMove**, **SimpleMove** y **OnCantMove**, ésta última es abstracta y cada clase heredada debe implementarla de la manera que considere. Las dos primeras básicamente van comprobando colisiones (con ayuda de Box Collider 2D) y si el objeto puede ir moviéndose en la dirección indicada o si, por el contrario, no puede, que llamaría a OnCantMove.

Ahora veamos cómo el usuario puede incidir en el movimiento, llega el turno de explicar los *VirtualJoystick*.

Vamos a comenzar explicando la clase ***VirtualJoystick***. Solamente dispone de un atributo, a parte de las imágenes, que son GameObjects. El atributo es de tipo Vector3, una clase usada para el manejo de vectores, y su nombre es *inputVector*, su cometido es guardar el vector que se está describiendo con el puntero. Es una clase que hereda de `IDragHandler`, `IPointerUpHandler` y `IPointerDownHandler`, los cuales son necesarios para el manejo de lo que va a ser nuestro puntero del *joystick*. Esta herencia nos permite usar los siguientes métodos.

• **OnDrag** toma la posición del puntero con respecto a la posición de la imagen de fondo para saber qué dirección se está queriendo describir. Teniendo eso, actualiza los valores de inputVector, indicándole los valores correspondientes en los ejes x e y.

• **OnPointerDown** se usa para describir qué se debe hacer cuando se ha pulsado el puntero, esta función llama directamente a OnDrag.

• **OnPointerUp** se usa para describir qué se debe hacer cuando se ha soltado el puntero de la pantalla. Lo que hace es devolver el *inputVector* a los valores del vector 0, además de mover la imagen del puntero al centro, donde empezó.

Después de estas funciones, que son heredadas y han sido sobrescritas, nuestro *VirtualJoystick* tiene dos funciones para consultar cómo está el *inputVector* en

un instante determinado. Estas funciones se llaman **Horizontal** y **Vertical**. Su función consiste simplemente en devolver el valor de su eje correspondiente. Estas funciones son las únicas usadas por parte del Player a la hora de moverse, así calcula la dirección a la que el joystick le indica.

Es el turno de *VirtualJoystickSkillshot*. Esta clase hereda de *VirtualJoystick*, por lo que conserva la mayor parte de su implementación. Una de las diferencias, la más importante es la variación a la función **OnPointerUp**. La variación consiste en que, en el momento que se suelta el puntero, en esa dirección en concreto, es cuando debería lanzarse la *SkillshotProjectile*, y justo después de que esto suceda, realizar el código del padre, como si de un *VirtualJoystick* normal se tratase. Una vez tenemos esa dirección capturada en el *inputVector,* ya sabemos que esa es la que tiene que tener la *SkillshotProjectile* en su viaje por el campo, así que recogemos con **Horizontal** y **Vertical** y la enviamos a Player para que genere la *SkillshotProjectile* correctamente.

Otra diferencia es que este *joystick* tiene una *Skillshot* asociada y su imagen se puede ver en el propio *joystick*, como vimos en la explicación de los componentes del HUD.

## 7.5. Gestión de las funcionalidades de red

Llegados a este punto, toca hablar sobre las funcionalidades de red que nos brinda NetworkBehaviour [50].

Antes de nada, vamos a explicar cómo realiza UNET [51] las acciones a través de la red. Estas acciones se conocen como Llamadas a Procedimiento Remoto (*Remote Procedure Calls o RPC)*. Hay dos tipos, los Comandos (*Commands)* que son llamados desde el cliente al servidor y las llamadas *ClientRPC*, que son llamadas en el servidor y ejecutadas en el cliente.

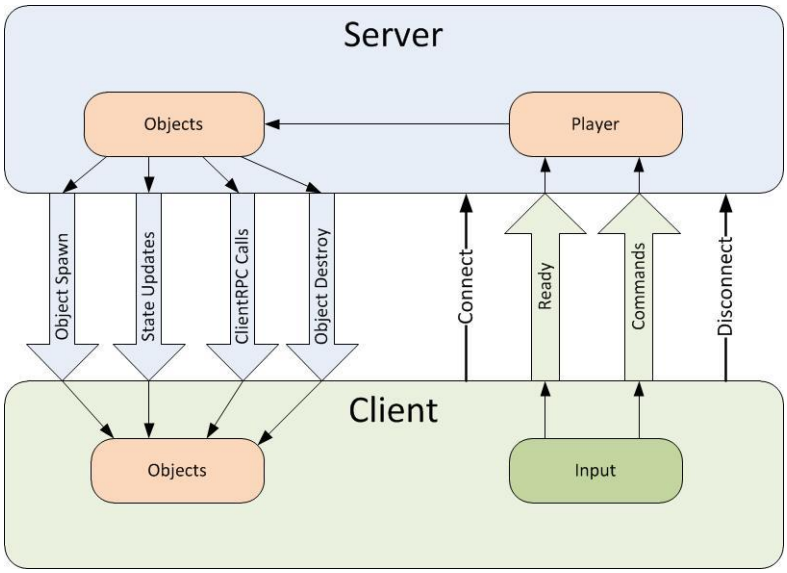


*Figura 7.31. Flujo de acciones Cliente-Servidor UNET*

En el diagrama anterior podemos identificar y diferenciar 4 elementos principales:

• El **Input**, que representa el "manejador" del cliente con su Player (que se encuentra en el servidor).

• El **Player** que es la instancia del personaje jugable del usuario en el servidor.

• Los **Objetos de Servidor** son las representaciones de los objetos que se encuentran en el servidor, una instancia de ellos.

• Los **Objetos de Cliente** son instancias copiadas de la instancia del servidor. Esta copia se realiza mediante el método *Spawn(gameObject)*, siendo *gameObject* el objeto que se encuentra en el servidor que queremos que aparezca en todos los clientes.

Como podemos ver, los comandos son llamadas por parte del cliente al servidor. Hacemos esto porque queremos que en el servidor se ejecute algo en concreto, lo veremos más adelante con detalle y ejemplos.

Por otro lado, si nos fijamos en los mensajes que van a de los objetos de servidor a los objetos de cliente, podemos observar, por ejemplo, *ObjectSpawn*, que sería lo mencionado anteriormente con el método *Spawn*, y Object*Destroy*, el cual es llamado cuando el objeto en el servidor es destruido, para que los de los clientes se destruyan también, manteniendo la consistencia en el juego. El resto de las llamadas las explicaremos más adelante.

### 7.5.1. Atributos

En C# podemos añadir unos atributos [52] a las variables y funciones, escribiendo el nombre del atributo por encima de la declaración de la variable entre corchetes, veremos un ejemplo más adelante. Estos atributos sirven de "etiqueta" para que otros fragmentos del código detecten estas variables o funciones y las traten de una forma especial. Gracias a NetworkBehaviour tenemos acceso a varios atributos que son realmente útiles para nuestro juego en red. A continuación, veremos los atributos que se han utilizado en este proyecto.

#### 7.5.1.1. SyncVar

Las variables que reciben el atributo *SyncVar* [53] son variables que están sincronizadas desde el servidor a los clientes. Cuando un GameObject es *spawneado* (instanciado en red), o un nuevo jugador se une a un juego en proceso, se les envía el último estado de las SyncVars de los objetos en red que son visibles para ellos. Las SyncVars solamente pueden afectar a variables de tipos básicos.

```
//Estadisticas
[SyncVar]
private float vida = 100;
[SyncVar]
private float vidaMaxima = 100;
[SyncVar]
private float ataque;
[SyncVar]
private float defensa;
[SyncVar]
private bool stunned = false;
[SyncVar]
private bool confuso = false;
```

*Figura 7.32. Ejemplos de atributos en variables*

Como podemos ver en la figura anterior, extraída del script de Player, hemos aplicado el atributo SyncVar a todas las estadísticas de éste. La razón es muy sencilla. Necesitamos que, en todo momento, todos los Players tengan exactamente igual y sincronizadas sus estadísticas. De otro modo, podría dar lugar a errores de desincronización.

Por ejemplo, Player1 tiene 70 puntos de ataque y recoge un objeto que le aumenta en 20, por tanto, tendría 90, y lanza una habilidad que impacta en Player2. Se realizará el cálculo de daño tanto en el dispositivo de Player1 como

en el dispositivo de Player2. Si todas estas estadísticas no estuvieran perfectamente sincronizadas, podría ocurrir que hubiera dos escenarios distintos sobre la misma partida, uno en cada dispositivo. En el de Player1, el Player2, con el cálculo de daño, podría haber muerto, en cambio, en el dispositivo de Player2, al contar con que Player1 solo tiene un ataque de 70, aún seguiría con vida, dando lugar a una situación incongruente y que podría arruinar la experiencia de juego. Por ese mismo motivo está sincronizada la vida.

Este atributo se correspondería con el mensaje *StateUpdates* del diagrama Cliente-Servidor.

#### 7.5.1.2. Command

Command es un atributo que pertenece a las funciones, con él, indicamos que la función afectada va a ser una llamada desde el **Input** al **Player** que está en el servidor. Por seguridad, los Comandos solo se pueden enviar desde el jugador local. Ya que los comandos están pensados para cambios de estado de parte del jugador en su propio **Player** y no tendría sentido que pudieras enviarlo desde otro Player en el dispositivo local que no fuera el tuyo propio. Para indicar que una función es un comando basta con poner el atributo de la misma manera que hacíamos con las SyncVar, y además, esa función debe tener el prefijo “Cmd”.

```
[Command]
public void CmdPlayerMuere(uint networkIdentity)
```

*Figura 7.33. Ejemplo de Command*

Como vemos en la figura anterior, podemos enviar parámetros con la llamada de la función, solamente necesitamos que tenga la sintaxis indicada. En el momento que en nuestro dispositivo local, desencadenemos los eventos que llamen a alguna función Command, la función se ejecutará en el servidor.

No es recomendable mandar Commands en cada frame. Están más bien orientados a eventos concretos, no a algo que se pueda repetir 50 veces por segundo. Esto se debe a que puede generar mucho tráfico de red, algo que no es recomendable.

#### 7.5.1.3. ClientRpc

ClientRpc es un atributo que, como Command, pertenece a las funciones. Sin embargo, esta vez, está orientada para las funciones que van desde los **Object del servidor** a los **Object del cliente**. Se pueden enviar desde cualquier objeto que esté en el servidor que posea un NetworkIdentity y que haya sido *spawneado* en todos los clientes. Básicamente, es una función que llama se llama en el servidor, en uno de sus objetos, para que esa función se realice en todas esas “copias” de los clientes.

```
[ClientRpc]
public void RpcMatarPlayer(uint networkIdentity)
```

*Figura 7.34. Ejemplo de ClientRpc*

Como vemos en la figura anterior, la sintaxis en el caso de este atributo es, “ClientRPC” como atributo y “Rpc” como prefjio del nombre de la función.

Tanto de Command, como de ClientRpc, aún nos quedan por explicar en qué momentos y para qué funciones los hemos utilizado. Se entenderá mejor su funcionamiento en ese apartado.

### 7.5.2. Variables de red

Al igual que los atributos, hay otras funcionalidades que contiene NetworkBehaviour, estas son las variables de red. Son variables que almacenan información importante sobre su GameObject. Veremos algunos de los ejemplos más importantes que hemos utilizado.

• isLocalPlayer

Es una variable booleana que indica, en el momento que la consultes, si éste Player sobre el que la estás consultando es el local. Útil para acciones que quieres que solo pasen en el dispositivo del jugador y no de manera global, por ejemplo, como comentamos anteriormente, la llamada de Commands.

• isServer

Variable booleana que indica, en el momento que la consultes, si este código se está ejecutando en el servidor. Útil para realizar ciertas acciones solamente una vez y en el servidor, por ejemplo, las llamadas de los RpcClient.

• netId

Variable que posee cada elemento que tiene un NetworkIdentity y que por tanto, está en el servidor. Cada elemento que está en el servidor recibe una identificación única que ayuda a gestionar de forma única cada elemento. Los objetos en servidor comparten *netId* con sus copias *spawneadas* en los clientes.

### 7.5.3. Uso de atributos y funciones en el proyecto

#### 7.5.3.1. Player

Como mencionamos en el apartado de las SyncVar, en Player todas las estadísticas tienen este atributo. En cuanto a los Commands y RpcClient, tenemos los siguientes:

• **CmdSpawnItem**, se llama de forma periódica y solo se ejecuta en el Player que a su vez es el que aloja la partida. Invoca un ítem y se lo muestra al resto de clientes.

• **CmdCrearSkillshotProjectile** se encarga de instanciar la *SkillshotProjectile*, darle dirección, rotación y el networkIdentity del Player que lo lanza, para que ese proyectil lleve la información del Player que lo lanzó, necesario en el cálculo de daño. Además, después de todo esto, llama a **RpcInicializarValores** una función de ***SkillshotProjectile*** que explicaremos más adelante.

• **CmdPlayerMuere** se llama cuando el jugador local ha detectado su propia muerte y desactiva (o hace desaparecer) al Player del campo. Además, al ser un Command y ejecutarse en servidor, se aprovecha para llamar a **RpcMatarPlayer**, cuyo funcionamiento explicaremos a continuación.

• **RpcMatarPlayer** se ejecuta en cada Player cliente para notificar al MatchManager de la muerte de un oponente, gracias al NetworkIdentity pasado por parámetro. Además de notificarlo al MatchManager para el posterior cálculo de posiciones, hace desaparecer al Player muerto de la escena.

#### 7.5.3.2. SkillshotProjectile

Esta clase, al igual que Player, también tiene ciertas variables afectadas con el atributo SyncVar. Estas son *xDirection* e *yDirection*.

En cuanto a los Commands y RpcClient, solo tiene un RpcClient bastante sencillo.

• **RpcInicializarValores** esta función cambiar ciertos valores del *SkillshotProjectile* como son la dirección y el Player lanzador de la habilidad. Esta función existe porque se debían inicializar estos valores justo después del *spawn* de estos en los dispositivos clientes, y es la forma en la que el servidor puede asegurarse de que estos valores cambian de la misma forma en todos los clientes.

### 7.5.4. Otras cuestiones derivadas de la red

Aunque no es propiamente una funcionalidad de red, estaría bien explicarlo en este apartado ya que su uso ha sido “obligado” a la hora de incorporar la red al juego. De lo que estamos hablando es de usar la función FixedUpdate en vez de Update en los elementos de la partida. Como vimos anteriormente, Update ejecuta su código cada vez que la pantalla se refresca. FixedUpdate en cambio, lo ejecuta a un ritmo constante, cada 0.02 segundos. Una problemática que ocurría al principio era que, a distintos dispositivos, el más potente (y por tanto más refrescos por unidad de tiempo) ejecutaba más veces el código que el que tenía peor dispositivo. Esto afectaba negativamente en la jugabilidad, ya que el que jugaba con el mejor móvil, su personaje daba la impresión de que corría más rápido. Efectivamente así ocurría, y no era un efecto visual, de falta de fluidez en uno o demasiada fluidez en otro. Sino que realmente, los “pasos” se estaban ejecutando más veces por unidad de tiempo en uno que en el otro, propiciando así que las distancias recorridas por unidad de tiempo fueran distintas. Para arreglar este problema hemos optado por usar FixedUpdate. Al especificar que el código que se ejecuta en cada refresco se va a hacer (tengas el dispositivo que tengas) cada 0.02 segundos, ninguno está ahora en desventaja, los “pasos” se ejecutan el mismo número de veces por unidad de tiempo, arreglando así el problema.

En cuanto al servidor utilizado, la contratación y administración es bastante sencilla, Unity nos facilita todo el proceso, por lo que simplemente se activa siguiendo un asistente y rellenando unos formularios dentro de su página. Una vez hecho esto, el videojuego tiene un servidor dedicado.

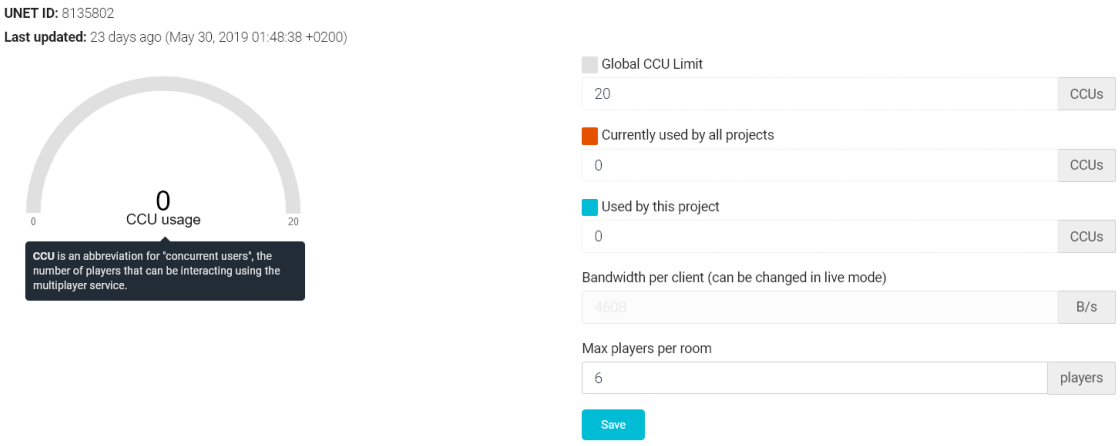


*Figura 7.35. Pantalla de administración que proporciona Unity*

## 7.6. Base de Datos con SQLite

En cuanto a la base de datos utilizada para la persistencia requerida por la aplicación, hemos decidido usar una solución sencilla, de bajo coste y que funcione bien en dispositivos móviles.


*Figura 7.36. Logo de SQLite*

SQLite [54] es una biblioteca en lenguaje C que implementa un motor de base de datos SQL pequeño, rápido, autónomo, de alta fiabilidad y completo. SQLite es el motor de base de datos más utilizado en el mundo. Está integrado en todos los teléfonos móviles y en la mayoría de las computadoras, y se incluye en innumerables aplicaciones que las personas usan todos los días.

En cuanto a la información que vamos a necesitar de forma persistente en la base de datos, se puede ver en el apartado del Modelo Relacional del capítulo de Diseño. Aun así, vamos a comentar cómo hemos implementado el acceso a esta base de datos.

Como también dijimos en el apartado de patrones de diseño, hemos aplicado un patrón Fachada, por lo que están centralizados todos los accesos, tanto para consulta, inserción y actualización en la misma clase de C#. Hemos llamado a esta clase SQLiteManager. A continuación, mencionaremos las funciones de las que disponemos en esta clase.

• **GetAllPlayerModifiers** devuelve todos los PlayerModifiers guardados en base de datos como objetos, incluidos sus arrays de SimpleModifiers.

• **GetAllPasivas** devuelve todos los PlayerModifiers que sean pasivas, en las mismas condiciones que la función anterior.

• **GetAllSkillshots** devuelve todos los *Skillshots*, incluido sus PlayerModifier, si tuvieran.

• **GuardarAbilitySets,** indicando el AbilitySet y el User, guarda la información actualizando la base de datos.

• **GuardarAbilitySet**, igual que el anterior pero solo para un AbilitySet.

• **InsertarPlayerModifier**, inserta un PlayerModifier en la base de datos, así como su array de SimpleModifier.

• **InsertarSkillshot**, inserta un *Skillshot* en la base de datos.

• **InsertarItem,** inserta un Item en la base de datos.

• **GetPlayerModifier,** indicando un índice, el idPlayerModifier, devuelve el PlayerModifier, con sus arrays de SimpleModifiers pertinentes.

• **GetSkillshot,** indicando un índice, el idSkillshot, devuelve el *Skillshot* y su PlayerModifier, si lo tuviera.

• **GetAbilitySets,** indicando un User, devuelve todos los AbilitySet que posee el usuario.

• **DarAltaUser,** indicando un User, crea el User y también la estructura del UserAbilitySet, con sus tres AbilitySet pertinentes.

• **UserAndUserAbilitySetExist** confirma, consultando un User, si ya existe, y si es así, también existe su UserAbilitySet.

Además de estas funciones, hay varias más dedicadas a la inicialización de las tablas, la limpieza total de las tablas y comprobar si las tablas están creadas. Son útiles para comprobar si las tablas ya han sido creadas y rellenadas con anterioridad, o por el contrario es la primera vez que se accede y hay que hacerlo, así como si el usuario está creado o no, y si no, darlo de alta. Todas estas gestiones se realizan desde una clase llamada **InicializarValoresBD**, que centraliza esta gestión, además de crear en ella los PlayerModifiers, *Skillshots* e Items que van a usarse en la partida.

## 7.7. Recursos utilizados

A la hora de crear un videojuego, como comentábamos al principio del capítulo, no basta con la programación de código. Ya vimos la parte relacionada con el editor de Unity creando y colocando los elementos de la escena, también vimos las animaciones. Pero hay elementos que ayudan a que el juego sea más vistoso, más agradable a la vista. Es el trabajo de los artistas y diseñadores gráficos. A menudo, estos brindan a la comunidad de desarrolladores sus creaciones bajo distintos tipos de licencias. Propiamente, en la misma tienda de Assets de Unity, podemos adquirir muchos elementos de este tipo. A continuación, haremos mención a los elementos que hemos adquirido y que ahora forman parte de nuestro proyecto.

### 7.7.1. Recursos adquiridos de la Asset Store de Unity

La Asset Store de Unity tiene multitud de elementos que puedes incorporar a tu proyecto, algunos incluso de forma gratuita. En nuestro caso, hemos adquirido los siguientes.

• **NetworkLobby**, compuesto por una serie de GameObject y clases que realizan las funciones de un Lobby. Permite al desarrollador que opte por incluir este *asset* en su proyecto, entrar en su código y modificarlo si se considera necesario. También proporciona la escena preparada con todos sus elementos de UI. Obviamente, esto también es personalizable.

*Figura 7.37. NetworkLobby*

En el caso de este proyecto, se realizó la funcionalidad en red completa apoyándose en un sistema de emparejamiento poco elegante. Era un sistema que usaba una clase que pertenece a la API de Unity llamada NetworkManager. Esta permite conectar a varios jugadores a disputar una partida mientras uno de ellos aloja la partida. Decimos que es poco elegante porque para poder conectarse a una partida, era necesario escribir de forma manual la IP del jugador que alojaba la partida cada vez que se quería jugar.

Una vez finalizada la funcionalidad completa y se comprobó, mediante las pruebas que veremos en el apartado correspondiente, que el juego funcionaba con normalidad, se decidió mejorar ese aspecto que nos parecía poco elegante. Por lo que se decidió pasar de ese sistema de emparejamiento a uno que, a ser posible, tuviera una manera más atractiva de interconectar a los usuarios. Así se descubrió este *asset* y se decidió iniciar la incorporación de él en nuestro proyecto.

Fue necesario informarse previamente sobre su funcionamiento, las clases que lo componían y cómo interactuaban entre sí, además de cómo tenían que interactuar con los elementos que ya funcionaban en nuestro juego. Después de algunos ajustes, tanto en el *asset* como en nuestras clases y escenas, y de la repetición de las pruebas, se pasó de la versión antigua a la actual, con el nuevo Lobby funcionando correctamente.

Enlace: https://assetstore.unity.com/packages/essentials/network-lobby-41836

• **Imágenes temáticas de distintos alimentos y bebidas**, al estilo Pixel Art, para dar vida a los *items* que van a ir apareciendo en el suelo.


*Figura 7.38. Pixel Food*

Enlace: https://assetstore.unity.com/packages/2d/environments/free-pixel-food-113523

## 7.7.2. Recursos adquiridos de distintos creadores

No solo de la Asset Store de Unity podemos conseguir elementos que incorporar a nuestro videojuego. Hay multitud de portales en los que la comunidad comparte sus diseños. Algunos de estos se ven sometidos a licencias, que cada autor indica en la página en la que puedes conseguir su aportación. Así pues, vamos a darle el crédito que se merecen a los creadores del contenido que hemos usado, bien para incluir tal cual o bien para modificarlo nosotros mismos y posteriormente incluirlo.

• **Avatar** para jugar la partida. Se buscaba un personaje, preferiblemente con aspecto de guerrero con espada, en 2D y con 8 direcciones. Se escogió finalmente este, que se ajustaba bastante bien a lo requerido.

Enlace: https://www.spriters-resource.com/sharp_x68000/thereturnofishtar/sheet/112403/

• **Fuente Pixelade** es la fuente que hemos utilizado para la mayoría de los menús y avisos dentro de la pantalla. Se buscaba una fuente con estilo pixel y esta es la idónea.

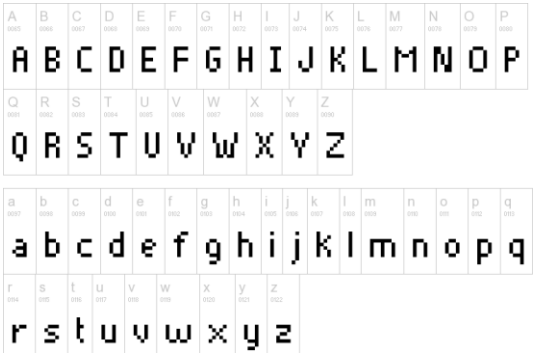


*Figura 7.39. Fuente Pixelade*

Enlace: https://www.dafont.com/es/pixelade.font

• **Imágenes para los Skillshot y los Player Modifiers.** Se requerían imágenes, con un aspecto no demasiado moderno, que representaran ciertas situaciones mágicas o de guerra. Hay un conocido juego de cartas, *Yu-gi-oh!*, el cual contiene muchas de esas situaciones y pensamos que podían quedar muy bien para estos dos cometidos.

Enlace:

https://www.spriters-resource.com/game_boy_advance/ygoeternalduelsoul/sheet/40049/

• **Imágenes para los menús y HUD.** Necesitábamos imágenes que nos pudieran servir, al menos como base, para realizar el HUD deseado y los menús acordes a la temática 2D pixel art. Encontramos la siguiente figura, y nos pareció que encajaba mucho con lo que queríamos.

*Figura 7.40. Imagen utilizada como base para el HUD*

Con respecto a esto, ha sido donde más hemos tenido que utilizar GIMP, la herramienta de edición. Usando como base esta imagen hemos podido editar hasta realizar los menús, los *joysticks*, botones y el resto de los elementos para que quedaran a nuestro gusto.

Enlace: https://opengameart.org/content/ui-pieces

• **Sprites para las SkillshotProjectile.** Necesitábamos varios Sprite o imágenes para realizar las animaciones que iban a acompañar a las *SkillshotProjectile* reproduciéndose en bucle durante todo su viaje por el campo de batalla. Estos son los que hemos usado, el resto han sido derivados de alguno de estos o bien creados totalmente de cero.

Enlaces:

https://opengameart.org/content/rotating-arrow-projectile (flecha normal)

https://opengameart.org/content/firebal-32x32 (bola de fuego)

https://opengameart.org/content/scimitar-sword (espada cuerpo a cuerpo)

# 8. Pruebas

En un desarrollo de software pueden aparecer errores en cualquier momento de su ciclo de vida, algunos de ellos puede que nunca se hayan materializado pero que permanezcan ocultos. Para evitar que, el día de mañana, nuestro desarrollo no sea defectuoso, es recomendable la realización de pruebas a lo largo de todo el proyecto.

En este capítulo presentaremos las pruebas realizadas como parte de la fase de transición del proyecto. El tipo de prueba elegido será caja negra. En cada prueba podremos ver los resultados obtenidos, así como las correcciones necesarias en caso de la ocurrencia de un fallo. Vamos a organizar las pruebas en dos grandes grupos.

## 8.1. Pruebas unitarias

Son las pruebas que confirman el correcto funcionamiento de un bloque de código en concreto. Al ser un videojuego, la mayor parte de las comprobaciones engloban varios casos de uso de una forma inevitable, ya que se tienen que realizar de forma manual, a través de eventos que provengan del usuario. Aun así, vamos a intentar separar las pruebas para que sean lo más unitarias posible, de manera que se asemejen a pequeñas situaciones que se deben cumplir en el juego. Dividiremos las pruebas en pruebas fuera de partida y dentro de la partida.

### 8.1.1. Pruebas unitarias fuera de la partida

#### 8.1.1.1. Identificarse

| PCN-001 | Identificación correcta sin usuario creado previamente |
|---|---|
| Descripción | El usuario se identifica en el sistema sin tener previamente ese nombre registrado en el dispositivo. |
| Resultado esperado | Se crea un nuevo usuario y el usuario queda identificado. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.1. PCN-001. Identificación correcta sin usuario creado previamente*

| PCN-002 | Identificación correcta con usuario creado previamente |
|---|---|
| Descripción | El usuario se identifica en el sistema teniendo previamente un usuario registrado en el dispositivo. |
| Resultado esperado | El sistema recupera de base de datos toda la información del usuario y éste queda identificado. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.2. PCN-002. Identificación correcta con usuario creado previamente*

### 8.1.1.2. Personalizar

| PCN-003 | Entrar correctamente al menú de Personalizar |
|---|---|
| Descripción | El usuario entra en Personalizar, pulsando su botón en el menú. |
| Resultado esperado | Al cargarse la nueva escena, aparecen correctamente sus AbilitySet con sus nombres y el seleccionado favorito correctamente. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.3. PCN-003. Entrar correctamente al menú de Personalizar*

| PCN-004 | Seleccionar un AbilitySet como favorito |
|---|---|
| Descripción | El usuario pulsa en el botón de la estrella del AbilitySet que desee hacer su favorito. |
| Resultado esperado | La estrella pulsada cambia a amarillo y la anterior favorita pasa a estar vacía. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.4. PCN-004. Seleccionar un AbilitySet como favorito*

| PCN-005 | Entrar a editar un AbilitySet |
|---|---|
| Descripción | El usuario pulsa en un AbilitySet para editarlo. |
| Resultado esperado | La nueva escena carga todas las Skillshot y Pasivas disponibles para el usuario, además de las que ya tenía guardadas en ese AbilitySet. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.5. PCN-005. Entrar a editar un AbilitySet*

| PCN-006 | Seleccionar una Skillshot |
|---|---|
| Descripción | El usuario pulsa en una Skillshot. |
| Resultado esperado | La información de la Skillshot se carga en el panel de información. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.6. PCN-006. Seleccionar una Skillshot*

| PCN-007 | Seleccionar una Pasiva |
|---|---|
| Descripción | El usuario pulsa en una Pasiva. |
| Resultado esperado | La información de la Pasiva se carga en el panel de información. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.7. Seleccionar una Pasiva*

| PCN-008 | Cambiar Skillshot/Pasiva del AbilitySet por la seleccionada |
|---|---|
| Descripción | El usuario pulsa en la ranura de Skillshot/Pasiva que desee, previamente de tener una seleccionada en el panel de información. |
| Resultado esperado | La ranura pasa a tener la imagen de la Skillshot/Pasiva elegida. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.8. PCN-008. Cambiar Skillshot/Pasiva del AbilitySet por la seleccionada*

| PCN-009 | Cambiar el nombre del AbilitySet |
|---|---|
| Descripción | El usuario pulsa en el AbilitySet e introduce el nuevo nombre. |
| Resultado esperado | El nombre del AbilitySet cambia correctamente. |
| Fase de aplicación | Fase final. |
| Valoración | OK |

*Tabla 8.9. PCN-009. Cambiar el nombre del AbilitySet*

| PCN-010 | Guardar toda la información cambiada |
|---|---|
| Descripción | El usuario pulsa el botón de guardar después de haber realizado cambios. |
| Resultado esperado | Los nuevos cambios quedan reflejados en la base de datos. |
| Fase de aplicación | Fase de desarrollo. |
| Valoración | Incorrecto. Hubo unos problemas con la base de datos relativos a que había errores en la sintaxis de SQLite. Después de varias depuraciones se consiguió que toda la sintaxis fuera la adecuada y todo funcionara correctamente. |

*Tabla 8.10. PCN-010. Guardar toda la información cambiada*

| PCN-011 | Guardar toda la información cambiada |
|---|---|
| Descripción | El usuario pulsa el botón de guardar después de haber realizado cambios. |
| Resultado esperado | Los nuevos cambios quedan reflejados en la base de datos. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.11. PCN-011. Guardar toda la información cambiada*

| PCN-012 | Salir sin guardar |
|---|---|
| Descripción | El usuario, tras haber realizado cambios, decide salir sin guardar la información cambiada. |
| Resultado esperado | El AbilitySet permanece como estaba antes de entrar a la edición. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.12. PCN-012. Salir sin guardar*

#### 8.1.1.3. Lobby

| PCN-013 | Pulsar el botón de Jugar |
|---|---|
| Descripción | El usuario pulsa el botón de jugar. |
| Resultado esperado | El sistema carga la escena del Lobby correctamente. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.13. PCN-013. Pulsar el botón de Jugar*

| PCN-014 | Crear partida |
|---|---|
| Descripción | El usuario crea una partida añadiendo un nombre. |
| Resultado esperado | El sistema crea la partida, el servidor la aloja y une al usuario a ella. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.14. PCN-014. Crear partida*

| PCN-015 | Buscar partidas creadas |
|---|---|
| Descripción | El usuario pulsa en el botón Buscar Partidas. |
| Resultado esperado | El sistema muestra un listado de las partidas disponibles alojadas en el servidor creadas por otros usuarios. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.15. PCN-015. Buscar partidas creada*

| PCN-016 | Unirse a una partida |
|---|---|
| Descripción | El usuario pulsa el botón de Unirse a una partida de otro usuario. |
| Resultado esperado | El sistema, junto con el servidor, une al usuario a la sala de la partida. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.16. PCN-016. Unirse a una partida*

| PCN-017 | Personalizar el color y el nombre antes de la partida |
|---|---|
| Descripción | El usuario personaliza el color y el nombre con el que jugará la partida. |
| Resultado esperado | El sistema guarda la información y se la muestra al resto de los usuarios, con ayuda del servidor. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.17. PCN-017. Personalizar el color y el nombre antes de la partida*

| PCN-018 | Pulsar el botón “Preparado” e inicia la partida |
|---|---|
| Descripción | El usuario pulsa el botón preparado y espera que el resto de los jugadores lo hagan. |
| Resultado esperado | El servidor espera a que todos pulsen el botón, entonces inicia la partida. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.18. PCN-018. Pulsar el botón de "Preparado" e inicia la partida*

## 8.1.2. Pruebas unitarias dentro de la partida

### 8.1.2.1. Pruebas con ámbito local

| PCN-019 | La partida empieza para el Player |
|---|---|
| Descripción | El jugador, que previamente pulsó el botón de preparado, ya se ha unido a la partida. |
| Resultado esperado | El usuario puede ver correctamente a su Player, el color y el nombre que ha elegido, así como su barra de vida encima de éste. También el HUD ha sido cargado correctamente con la configuración elegida del AbilitySet favorito. |
| Fase de aplicación | Fase de desarrollo |
| Valoración | Incorrecto. Hubo varios fallos a la hora de cuadrar bien el HUD con el Player y que se mostraran todos y cada uno de los componentes de forma adecuada. Algunos de los problemas venían derivados de la utilización de Start en vez de Awake, y al usar esta última se arreglaban. Esto era porque Awake se ejecuta antes que cualquier Start de cualquier Behaviour, y a veces era necesario que así lo haciera. |

*Tabla 8.19. PCN-019. La partida empieza para el Player*

| PCN-020 | La partida empieza para el Player |
|---|---|
| Descripción | El jugador, que previamente pulsó el botón de preparado, ya se ha unido a la partida. |
| Resultado esperado | El usuario puede ver correctamente a su Player, el color y el nombre que ha elegido, así como su barra de vida encima de éste. También el HUD ha sido cargado correctamente con la configuración elegida del AbilitySet favorito. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.20. PCN-020. La partida empieza para el Player*

| PCN-021 | El HUD se mantiene actualizado en todo momento |
|---|---|
| Descripción | El sistema debe mantener actualizado el HUD en todo momento, reflejando el estado del Player. Para probar esto, se realiza cualquier escenario en el que el HUD tenga que ser actualizado, como recoger un objeto del suelo. |
| Resultado esperado | El HUD se mantiene actualizado correctamente en todo momento. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.21. PCN-021. El HUD se mantiene actualizado en todo momento*

| PCN-022 | El usuario mueve al Player con el Joystick |
|---|---|
| Descripción | El usuario usa el joystick con la intención de mover al usuario. |
| Resultado esperado | El Player se mueve en la dirección indicada en el joystick. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.22. PCN-022. El usuario mueve al Player con el Joystick*

| PCN-023 | Lanzar Skillshot |
|---|---|
| Descripción | El usuario puede lanzar una Skillshot con ayuda de cualquiera de los joysticks izquierdos. |
| Resultado esperado | Al soltar al joystick, la habilidad sale de la ubicación del Player en la dirección y orientación adecuados. |
| Fase de aplicación | Fase de desarrollo |
| Valoración | Incorrecto. Hubo algún fallo con la orientación de esta a la hora de lanzar, pero se solucionó. Otro de ellos era que entraba en colisión con el propio Player, al aparecer justo en el mismo punto que el lanzador. Esto se solucionó calculando, junto con la dirección que iba a tomar la habilidad, un punto a partir del Player en esa dirección, a una distancia prudencial para que no entrara en colisión y pudiera salir correctamente. |

*Tabla 8.23. PCN-023. Lanzar Skillshot*

| PCN-024 | Lanzar Skillshot |
|---|---|
| Descripción | El usuario puede lanzar una Skillshot con ayuda de cualquiera de los joysticks izquierdos. |
| Resultado esperado | Al soltar al joystick, la habilidad sale de la ubicación del Player en la dirección y orientación adecuados. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.24. PCN-024. Lanzar Skillshot*

| PCN-025 | Recoger objeto |
|---|---|
| Descripción | El usuario mueve al Player por encima de la posición de un objeto que se encuentre en el suelo. |
| Resultado esperado | El objeto desaparece. |
| Fase de aplicación | Fase Final |
| Valoración | OK |

*Tabla 8.25. PCN-025. Recoger objeto*

| PCN-026 | El Player ve afectadas sus estadísticas al recoger un objeto |
|---|---|
| Descripción | Después de recoger un objeto, el Player ve afectadas sus estadísticas. |
| Resultado esperado | Las estadísticas han sido afectadas correctamente. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.26. PCN-026. El Player ve afectadas sus estadísticas al recoger un objeto*

| PCN-027 | Se revierten los efectos aplicados por un PlayerModifier |
|---|---|
| Descripción | Al cabo de un tiempo, la afección derivada por un PlayerModifier, ya sea por objeto o por una Skillshot, se ve revertida. |
| Resultado esperado | La afección se revierte correctamente. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.27. PCN-027. Se revierten los efectos aplicados por un PlayerModifier*

| PCN-028 | Muerte de un Player |
|---|---|
| Descripción | Si la vida del Player llega a cero, desaparece y muestra al usuario un mensaje de fin de partida. |
| Resultado esperado | Cuando la vida del Player baja de cero debe mostrar el mensaje concreto de la posición en la que ha quedado el jugador y finalizar la partida para él. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.28. PCN-028. Muerte de un Player*

| PCN-029 | La partida llega al tiempo límite |
|---|---|
| Descripción | El tiempo límite de la partida ha llegado. |
| Resultado esperado | El sistema debería dar por acabada la partida y mostrar al usuario un mensaje de ello. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.29. PCN-029. La partida llega al tiempo límite*

### 8.1.2.2. Pruebas con ámbito de red

| PCN-030 | Correcto funcionamiento de las posiciones de los jugadores en red |
|---|---|
| Descripción | Al moverse otro usuario, se ve su movimiento y animación de movimiento correctamente en la pantalla local. |
| Resultado esperado | Las posiciones de los jugadores en red están sincronizadas en todos los dispositivos. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.30. PCN-030. Correcto funcionamiento de las posiciones de los jugadores en red*

| PCN-031 | Correcto funcionamiento de las Skillshot en red |
|---|---|
| Descripción | Si otro usuario lanza una Skillshot, se ve su movimiento y orientación en la pantalla local. |
| Resultado esperado | La posición y orientación de todas las Skillshots están sincronizadas en todos los dispositivos. |
| Fase de aplicación | Fase de desarrollo |
| Valoración | Incorrecto. Las skillshot funcionaban bien en local pero no aparecían en red. Hubo que replantear la forma de implementarlas y finalmente se solucionó el problema. |

*Tabla 8.31. PCN-031. Correcto funcionamiento de las Skillshot en red*

| PCN-032 | Correcto funcionamiento de las Skillshot en red |
|---|---|
| Descripción | Si otro usuario lanza una Skillshot, se ve su movimiento y orientación en la pantalla local. |
| Resultado esperado | La posición y orientación de todas las Skillshots están sincronizadas en todos los dispositivos. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.32. PCN-032. Correcto funcionamiento de las Skillshot en red*

| PCN-033 | Correcto funcionamiento de la recogida de objetos en red |
|---|---|
| Descripción | Si otro usuario recoge ese objeto, desaparece de la pantalla local. |
| Resultado esperado | El objeto desaparece de todas las pantallas locales y el Player del otro usuario ve alteradas correctamente sus características. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.33. PCN-033. Correcto funcionamiento de la recogida de objetos en red*

| PCN-034 | Muerte de otro Player |
|---|---|
| Descripción | La vida de otro jugador, que no es el local, llega a cero. |
| Resultado esperado | Ese Player desaparece de la pantalla local y se le notifica al MatchManager de que hay un jugador menos, para que lo tenga en cuenta a la hora de calcular las posiciones. |
| Fase de aplicación | Fase de desarrollo |
| Valoración | Incorrecto. En ocasiones no funcionaba bien y en la pantalla local moría, pero en el resto no desaparecía o viceversa. Se llegó a la solución, que resultaron ser una secuencia de errores simples que no dejaban llegar el flujo de ejecución hasta la llamada al RpcMatarPlayer y también se depuró la forma en la que MatchManager calcula las posiciones, que también daba algún error. |

*Tabla 8.34. PCN-034. Muerte de otro Player*

| PCN-035 | Muerte de otro Player |
|---|---|
| Descripción | La vida de otro jugador, que no es el local, llega a cero. |
| Resultado esperado | Ese Player desaparece de la pantalla local y se le notifica al MatchManager de que hay un jugador menos, para que lo tenga en cuenta a la hora de calcular las posiciones. |
| Fase de aplicación | Fase final |
| Valoración | OK |

*Tabla 8.35. PCN-035. Muerte de otro Player*

| PCN-036 | Player impactado por una Skillshot |
|---|---|
| Descripción | Una Skillshot lanzada por un usuario impacta en otro. |
| Resultado esperado | En todas las pantallas locales debería verse correctamente cómo ese Player ha recibido el daño basado en el ataque del lanzador y la defensa del receptor. Y se ve afectado por un PlayerModifier en caso de que esa Skillshot tuviera. |
| Fase de aplicación | Fase de desarrollo |
| Valoración | Incorrecto. La habilidad se quedaba volando contra el Player en vez de impactarle y desaparecer. Posteriormente se arregló el problema cambiando el modo en el que el Collider del Player funcionaba. |

*Tabla 8.36. PCN-036. Player impactado por una Skillshot*

| PCN-037 | Player impactado por una Skillshot |
|---|---|
| **Descripción** | Una Skillshot lanzada por un usuario impacta en otro. |
| **Resultado esperado** | En todas las pantallas locales debería verse correctamente cómo ese Player ha recibido el daño basado en el ataque del lanzador y la defensa del receptor. Y se ve afectado por un PlayerModifier en caso de que esa Skillshot tuviera. |
| **Fase de aplicación** | Fase final |
| **Valoración** | OK |

*Tabla 8.37. PCN-037. Player impactado por una Skillshot*

## 8.2. Pruebas de integración

Las pruebas de integración son las que se realizan una vez se han comprobado las pruebas unitarias, así comprobamos si todos los componentes que hemos probado individualmente funcionan correctamente en conjunto. Haremos tres pruebas de integración, para intentar agrupar y explotar el conjunto de las pruebas unitarias en sus respectivos ámbitos.

### 8.2.1. Prueba de integración en Personalizar

En escenario se centrará en intentar explotar todas las posibilidades relativas a Personalizar.

| PI-001 | Prueba de integración de los menús de personalizar |
|---|---|
| **Caso de prueba:** | 1. Pulsar Personalizar.<br>2. Entrar a editar un AbilitySet.<br>3. Cambiar todas las ranuras tanto de Skillshot como de Pasivas.<br>4. Cambiar el nombre.<br>5. Guardar.<br>6. Salir hacia atrás.<br>7. Marcar como favorito el AbilitySet editado. |
| **Fase de aplicación** | Fase final |
| **Valoración** | OK |

*Tabla 8.38. PI-001. Prueba de integración de los menús de personalizar*

### 8.2.2. Prueba de integración en Lobby

En este escenario nos centraremos en realizar los dos posibles casos de prueba; el ser el creador de una partida y esperar que alguien más se una para dar comienzo a la partida y el ser el usuario que se una a la sala ya creada.

| PI-002 | Prueba de integración para el creador de partida |
|---|---|
| **Caso de prueba:** | 1. Pulsar en Jugar.<br>2. Añadir un nombre de partida y pulsar “Crear”.<br>3. Personalizar nombre y color y esperar a otro usuario.<br>4. Pulsar botón de “Preparado”. |
| **Fase de aplicación** | Fase final |
| **Valoración** | OK |

*Tabla 8.39. PI-002. Prueba de integración para el creador de la partida*

| PI-003 | Prueba de integración para unirse a una partida |
|---|---|
| **Caso de prueba:** | 1. Pulsar en Jugar.<br>2. Pulsar en “Buscar partidas”.<br>3. Unirse a una partida creada.<br>4. Personalizar nombre y color.<br>5. Pulsar botón de “Preparado”. |
| **Fase de aplicación** | Fase final |
| **Valoración** | OK |

*Tabla 8.40. PI-003. Prueba de integración para unirse a una partida*

### 8.2.3. Prueba de integración en partida

Este escenario consistirá en probar a jugar una partida con más de un usuario, haciendo que los usuarios que estén probando utilicen todos los mecanismos que les brinda el juego para probarlo en su totalidad.

| **PI-004** | **Prueba de integración en partida** |
|---|---|
| **Caso de prueba:** | • Al menos debería recoger un objeto cada Player.<br>• Deberían lanzar al menos una vez cada una de sus habilidades, impactando y sin impactar en el contrario.<br>• Deberían moverse por todo el mapa libremente.<br>• Debería acabar la partida y mostrar la posición que ha conseguido cada uno. |
| **Fase de aplicación** | Fase final |
| **Valoración** | OK |

*Tabla 8.41. PI-004. Prueba de integración en partida*

# 9. Conclusiones

Para finalizar este Trabajo de Fin de Grado, nos gustaría exponer las conclusiones que hemos obtenido a raíz de la realización de este. Entraremos a valorar cuestiones tanto del proceso realizado como de los objetivos conseguidos, así como de las futuras líneas de desarrollo que podrían seguirse si se quisiera ampliar el proyecto.

## 9.1. Conclusiones generales

Desde el inicio del proyecto, siempre se han tenido en mente los objetivos hasta su consecución. Hemos comprendido el proceso de desarrollo de un videojuego desde cero. Hemos sido capaces de adaptar los conocimientos adquiridos durante la carrera en las fases de análisis diseño e implementación. Hemos realizado la planificación de un proyecto grande y aprendido de nuestros errores con respecto a las estimaciones. Hemos alcanzado las expectativas propuestas en materia de aprendizaje del motor de videojuegos Unity y de sus funcionalidades en red para poder aplicarlas a nuestro proyecto. Hemos podido observar cómo otras compañías líderes en el sector han resuelto los problemas planteados en sus productos y les hemos tomado como referencia para aplicar nuestras soluciones.

Hemos conseguido todo esto, hemos sido capaces de desarrollar un juego que cumple con todos sus requisitos iniciales a la par que entretiene. Gracias a los conocimientos que ya teníamos de la carrera, hemos podido afrontar de una forma adecuada el reto que teníamos, haciendo que, de forma colateral, afianzáramos estos valores mucho mejor.

Al haber sido este proyecto, hasta la fecha, el más grande que se ha realizado por parte del equipo, nos ha permitido valorar realmente el trabajo de cada una de las fases que conlleva un producto software de tal envergadura como es un videojuego.

## 9.2. Futuras líneas de desarrollo

Al ser un juego de invención propia, en su inicio había una gran cantidad de ideas relacionadas con el juego y todas sus posibilidades. Al disponer de recursos limitados, se decidió centrar el trabajo en lo que actualmente es, un juego multijugador que permitiera a los usuarios personalizar todo antes de entrar a jugar, y que pudieran hacerlo desde cualquier punto del planeta. Eso fue lo que se decidió, pero hubo varias ideas que se quedaron pendientes y se podrían aplicar en una posible continuación de este proyecto. A continuación, mencionaremos algunas de ellas:

• Inclusión de elementos de audio:

Daría un valor añadido añadir pequeños elementos sonoros que hicieran más envolvente y atractivo al usuario el videojuego. Elementos como una música de fondo para los menús o pequeños sonidos cada vez que ocurriera algo en partida.

• Sistema de niveles de los usuarios:

Con vistas a que el juego fuera lanzado para su comercialización, sería un componente que motivaría a los usuarios el tener una progresión por niveles de su usuario. También se pensó, que, con el aumento de niveles, se podrían ir desbloqueando tanto skillshots como pasivas para el usuario, teniendo así un motivo para que continúen jugando motivados.

• Sistema de clasificación:

Siguiendo con la tónica del punto anterior, un sistema de clasificación por ligas haría que los usuarios se motivaran más en mejorar al retarse con sus amigos por ver quién llega más alto. También se implantaría un sistema de puntuación al finalizar la partida para ello.

Todas estas ideas no terminaron formando parte del proyecto finalmente porque las hemos considerado añadidos. Todos orientados a motivar a los usuarios y a

mantenerlos fieles al juego. Pero realmente no nos resultaban relevantes para el objetivo inicial del proyecto, por lo que se decidió centrarse en él.

# Bibliografía

[1] «AEVI,» [En línea]. Available: http://www.aevi.org.es/la-industria-del-videojuego/en-el-mundo/. [Último acceso: 20 06 2019].

[2] D. E. Comer, Internetworking with TCP/IP - Principles, Protocols and Architecture (Sexta Edición), Pearson, 2014.

[3] J. Ray, TCP/IP (Edición Especial), Prentice Hall, 1998.

[4] D. E. &. S. D. L. Comer, Interconectividad de redes con TCP/IP, Prentice Hall, 2000.

[5] F. Halsall, Redes de Computadores e Internet (Quinta edición), Pearson, 2006.

[6] Varios autores, Guía completa de protocolos de telecomunicaciones, McGraw Hill, 2002.

[7] «Fuente de Figura,» [En línea]. Available: https://i2.wp.com/redes.noralemilenio.es/wp-content/uploads/2015/04/Cabecera-TCP.jpg. [Último acceso: 20 06 2019].

[8] «Fuente de Figura,» [En línea]. Available: http://www.ni.com/cms/images/devzone/tut/UDP%20Datagram%20Picture.JPG. [Último acceso: 20 06 2019].

[9] CNMC Blog, «CNMC Blog,» 24 01 2017. [En línea]. Available: https://blog.cnmc.es/2017/01/24/prepago-en-declive/. [Último acceso: 20 06 2019].

[10] G. Cuadrado, «El Androide libre,» 02 02 2017. [En línea]. Available: https://elandroidelibre.elespanol.com/2017/02/evolucion-tarifas-datos.html. [Último acceso: 20 06 2019].

[11] «Ditrendia,» 02 2018. [En línea]. Available: http://mktefa.ditrendia.es/blog/todas-las-estadísticas-sobre-móviles-que-deberías-conocer-mwc18. [Último acceso: 20 06 2019].

[12] «WikiBooks,» [En línea]. Available: https://es.wikibooks.org/wiki/Creación_de_videojuegos/Breve_historia_de_los_videojuegos. [Último acceso: 20 06 2019].

[13] «Wikipedia - PLATO,» [En línea]. Available: https://es.wikipedia.org/wiki/Programmed_Logic_Automated_Teaching_Operations. [Último acceso: 20 06 2019].

[14] «Wikipedia - Spasim,» [En línea]. Available: https://es.wikipedia.org/wiki/Spasim. [Último acceso: 20 06 2019].

[15] «Fuente de Figura,» [En línea]. Available: https://www.youtube.com/watch?v=yjOvL_QsTzA. [Último acceso: 20 06 2019].

[16] «Wikipedia - Half-Life,» [En línea]. Available: https://es.wikipedia.org/wiki/Half-Life. [Último acceso: 20 06 2019].

[17] «Fuente de Figura,» [En línea]. Available: https://cdns.kinguin.net/media/category/1/-/1-1024_844.jpg. [Último acceso: 20 06 2019].

[18] «Wikipedia - Counter-Strike,» [En línea]. Available: https://en.wikipedia.org/wiki/Counter-Strike. [Último acceso: 20 06 2019].

[19] «Fuente de Figura,» [En línea]. Available: https://cdn.wccftech.com/wp-content/uploads/2019/02/Super-Smash-Bros.-Ultimate-2-0-1.jpg. [Último acceso: 20 06 2019].

[20] «Fuente de Figura,» [En línea]. Available: https://www.hobbyconsolas.com/guias-trucos/fortnite/mejores-nuevas-zonas-lootear-fortnite-battle-royale-186020. [Último acceso: 20 06 2019].

[21] «Fuente de Figura,» [En línea]. Available: https://www.youtube.com/watch?v=UyWH5KhESZk. [Último acceso: 20 06 2019].

[22] «Fuente de Figura,» [En línea]. Available: https://media.redadn.es/imagenes/medieval-ii-total-war-kingdoms_123196.jpg. [Último acceso: 20 06 2019].

[23] «Fuente de Figura,» [En línea]. Available: https://i.ytimg.com/vi/xSwX1wzg458/maxresdefault.jpg. [Último acceso: 20 06 2019].

[24] «Wikipedia - World of Warcraft,» [En línea]. Available: https://es.wikipedia.org/wiki/World_of_Warcraft. [Último acceso: 20 06 2019].

[25] «Fuente de Figura,» [En línea]. Available: https://www.youtube.com/watch?v=MQ2UfR9yKl0. [Último acceso: 20 06 2019].

[26] «Wikipedia - League Of Legends,» [En línea]. Available: https://es.wikipedia.org/wiki/League_of_Legends. [Último acceso: 20 06 2019].

[27] Movistar eSports, «esports.as.com,» 20 03 2018. [En línea]. Available: https://esports.as.com/industria/esports_0_1117988194.html. [Último acceso: 20 06 2019].

[28] «Fuente de Figura,» [En línea]. Available: http://as01.epimg.net/esports/imagenes/2016/10/28/league_of_legends/1477682122_921805_1477683526_noticia_normal.jpg. [Último acceso: 20 06 2019].

[29] A. Merino, «Xataka,» 21 12 2017. [En línea]. Available: https://esports.xataka.com/lol-league-of-legends-1/el-league-of-legends-termina-el-ano-2017-con-unas-cifras-de-record. [Último acceso: 20 06 2019].

[30] M. Geig, Unity Game Development in 24 hours, Pearson, 2014.

[31] Adrián Domínguez, Fernando Navarro & Javier M. Castro, Unity 2017.x Curso Práctica, Ra-Ma, 2017.

[32] E. Pardos, «Baboonlab,» 2017. [En línea]. Available: http://www.baboonlab.com/blog/noticias-de-marketing-inmobiliario-y-tecnologia-1/post/unreal-engine-4-el-motor-grafico-que-ofrece-realismo-al-maximo-23. [Último acceso: 20 06 2019].

[33] A. Abramychev, «Unity Blog,» 11 06 2014. [En línea]. Available: https://blogs.unity3d.com/es/2014/06/11/all-about-the-unity-networking-transport-layer/. [Último acceso: 20 06 2019].

[34] E. D. Audiovisual, «eldocumentalistaudiovisual,» 06 02 2015. [En línea]. Available: https://eldocumentalistaudiovisual.com/2015/02/06/documentacion-en-videojuegos-documento-de-diseno-gdd/. [Último acceso: 20 06 2019].

[35] R. Burke, Project Management: Planning and Control Techniques (Cuarta edición), John Wiley & Sons, 2011.

[36] C. S. Dionisio, A project manager's book of forms (Tercera edición), John Wiley & Sons, 2017.

[37] N. Figuerola, «ArticulosPM,» 06 2015. [En línea]. Available: https://articulospm.files.wordpress.com/2015/06/riesgos-plan-mitigacion-vs-plan-contingencia-vs-fallback-plan.pdf. [Último acceso: 20 06 2019].

[38] «Fuente de Figura,» [En línea]. Available: https://mcsdlopez.files.wordpress.com/2012/12/rup_espanol.gif. [Último acceso: 20 06 2019].

[39] R. L. Martin, «UVA Docs,» 2018. [En línea]. Available: http://uvadoc.uva.es/handle/10324/33218. [Último acceso: 20 06 2019].

[40] I. Sommerville, Ingeniería del Software.

[41] «Fuente de Figura,» [En línea]. Available: http://uvadoc.uva.es/handle/10324/13198. [Último acceso: 20 06 2019].

[42] R. M. Vázquez, «Uva Docs,» [En línea]. Available: http://uvadoc.uva.es/handle/10324/27512. [Último acceso: 20 06 2019].

[43] Unity, «Unity3D - EventSystem,» [En línea]. Available: https://docs.unity3d.com/es/current/Manual/EventSystem.html. [Último acceso: 20 06 2019].

[44] Aldea Lúdica, «Aldea Lúdica,» [En línea]. Available: http://aldealudica.cerojugadores.com/edc/christopher-alexander/. [Último acceso: 20 06 2019].

[45] R. Nystrom, Game Programming Patterns, Lightning Source, 2014.

[46] R. Carr, «BlackWasp,» 22 08 2009. [En línea]. Available: http://www.blackwasp.co.uk/gofpatterns.aspx. [Último acceso: 20 06 2019].

[47] J. L. G. Sánchez, «Universidad de Granada - Jugabilidad como Calidad de la Experiencia del Jugador,» [En línea]. Available: http://lsi.ugr.es/juegos/articulos/interaccion09-jugabilidad.pdf. [Último acceso: 20 06 2019].

[48] J. L. G. Sánchez, «Universidad de Granada - Jugabilidad como Calidad de la Experiencia del Jugador,» [En línea]. Available: http://lsi.ugr.es/~juegos/articulos/interaccion08-jugabilidad.pdf. [Último acceso: 20 06 2019].

[49] «Fuente de Figura,» [En línea]. Available: https://www.geogebra.org/m/tezrgs9Z. [Último acceso: 20 06 2019].

[50] Unity, «Unity3D - NetworkBehaviour,» [En línea]. Available: https://docs.unity3d.com/Manual/class-NetworkBehaviour.html. [Último acceso: 20 06 2019].

[51] Unity, «Unity3D - UNET Actions,» [En línea]. Available: https://docs.unity3d.com/Manual/UNetActions.html. [Último acceso: 20 06 2019].

[52] Microsoft - Colaboradores, «Microsoft - Uso de atributos en C#,» [En línea]. Available: https://docs.microsoft.com/es-es/dotnet/csharp/tutorials/attributes. [Último acceso: 20 06 2019].

[53] Unity, «Unity3D - UNetStateSync,» [En línea]. Available: https://docs.unity3d.com/Manual/UNetStateSync.html. [Último acceso: 20 06 2019].

[54] SQLite, «SQLite,» [En línea]. Available: https://www.sqlite.org. [Último acceso: 20 06 2019].

[55] «Fuente de Figura,» [En línea]. Available: https://www.youtube.com/watch?v=MQ2UfR9yKl0. [Último acceso: 20 06 2019].

# Anexos

## A. Contenido del CD

El contenido del CD consta de:

Carpeta “Memoria”, contiene la memoria del TFG.

Carpeta “Fuentes del proyecto”, contiene los archivos fuente del proyecto de Unity.

Carpeta “Instalación del proyecto”, contiene el archivo “BattleInTheSand.apk”, el instalable del juego.

Carpeta “Manuales”, contiene tanto el manual de usuario, como el de instalación.

# B. Manual de instalación

Para instalar nuestra aplicación en un móvil con Android, lo primero que tenemos que hacer, en caso de que no lo esté, es ir a Ajustes de Android y después en el apartado Seguridad. Allí lo que debemos hacer es permitir la instalación de “aplicaciones de terceros” o de “orígenes desconocidos”. La configuración exacta de cada dispositivo y versión de Android puede diferir en esto, por lo que se recomienda buscar en internet un tutorial específico para nuestro dispositivo y que nos ayude a desbloquear esta opción.

Una vez activado esto, simplemente tenemos que descargar el archivo “BattleInTheSand.apk”, hacer click en él desde el explorador de archivos e instalarlo.

# C. Manual de usuario

A continuación, vamos a presentar el manual de usuario, el contenido seg el flujo habitual (que no necesario) para jugar correctamente a nuestro videojuego.

## Identificarse

La primera pantalla que nos aparece es la de identificarse, en la cual simplemente tendremos que añadir nuestro nombre de usuario.


*Figura 41. Identificarse*

En caso de no tener ningún perfil anterior con ese nombre, se creará uno nuevo. Por otro lado, si ya teníamos un perfil con ese nombre, nos cargará la información guardada que dejamos en la sesión anterior.

## Menú Principal

La siguiente figura muestra nuestro menú principal, desde el cual podemos elegir entre jugar directamente o, primero, personalizar nuestros sets de habilidades.


*Figura 42. Menú Principal*

En nuestro caso vamos a optar por personalizar antes de entrar a jugar.

## Personalizar

La siguiente figura es la que nos vamos a encontrar al pulsar el botón Personalizar. Cada uno de esos elementos representa un set de habilidades, y la estrella que tienen a su lado representa cuál de ellos es nuestro favorito, o lo que es lo mismo, el elegido para jugar.


*Figura 43. Personalizar*

Vamos a pulsar, por ejemplo, en AbilitySet 2 y entraremos a editarlo.

## Personalizar Set de Habilidades

Un set de habilidades consta de 3 habilidades pasivas y 3 habilidades lanzables o *skillshots*. Las habilidades pasivas son habilidades que no podremos usar durante la partida, sin embargo, le proporcionará a nuestro personaje una serie de modificaciones en sus atributos básicos. Las skillshots son habilidades lanzables que el usuario puede usar contra otros jugadores a modo de ataque.


*Figura 44. Edición del set de habilidades*

Vamos a explicar lo que podemos ver en la figura anterior, de izquierda a derecha y de arriba abajo:

• El botón para volver atrás.

• El nombre actual del set de habilidades, podemos pulsar sobre él para cambiar el nombre.

• El botón de guardar. Al pulsar en él guardaremos de forma permanente los cambios que hayamos realizado.

• Panel de información de la habilidad pasiva y/o de skillshot, que contiene una imagen representativa además de una breve descripción de su cometido.

• Botones para cambiar la interfaz entre Pasiva y Skillshot, dependiendo del botón que pulsemos nos mostrará todas las pasivas o todas las skillshots.

• Panel en el que seleccionar las pasivas/skillshots y poder así cargar su información en el panel correspondiente.

• Panel de ranuras disponibles para las pasivas y skillshots. Dependiendo también de la interfaz en la que nos encontremos, podremos asignar a nuestro set de habilidades la pasiva/skillshot en cuestión que se encuentre seleccionada en el panel de información.

Para empezar, podemos pulsar en la pestaña de Skillshots, y después pulsar en una al azar, por ejemplo, el Disparo a distancia.


*Figura 45. Seleccionando "disparo a distancia"*

Acto seguido, podemos pulsar en la ranura circular más a la derecha para asignar esa skillshot a esa posición.


*Figura 46. Disparo a distancia asignado*

Podemos probar también a seleccionar el bumerán y ponerlo en el que anteriormente hay una espada.


*Figura 47. Bumerán asignado*

Una vez hayamos personalizado las skillshots, llega la hora de las pasivas, que funciona exactamente igual. Vamos a colocar las tres primeras del panel en las tres ranuras que tenemos.


*Figura 48. Pasivas nuevas asignadas*

Pulsamos guardar y salimos al menú de Personalizar.


*Figura 49. Personalizar con SetPrueba*

Como podemos ver, el nombre del segundo set de habilidades ha cambiado, ahora vamos a seleccionarlo como favorito.


*Figura 50. Personalizar con SetPrueba como favorito*

Posteriormente, volveremos al menú y esta vez pulsaremos en Jugar.

## Vestíbulo de encontrar partida

Nos encontramos en el vestíbulo para encontrar partida. Desde aquí podemos tanto crear partida como encontrar una que haya creado otro usuario. Vamos a probar a crear una.


*Figura 51. Nombre de partida introducido, a punto de pulsar en Crear*

Introducimos el nombre y pulsamos en crear. Nos dirige a la pantalla siguiente, en la cual podemos personalizar el color que queremos que tenga nuestro personaje en partida y el nombre. Vamos a llamarnos “JugadorDePrueba” y a elegir el color verde, mientras esperamos que alguien más se una a la partida.


*Figura 52. Nombre y color personalizados para la partida*

Como podemos observar, otro usuario se ha unido a nuestra partida. Ahora tenemos que pulsar el botón que indica que estamos listos. Una vez pulsado, solo nos falta esperar a que el otro usuario haga lo mismo.


*Figura 53. Esperando a que el otro jugador esté preparado*

Una vez que todos los jugadores estén preparados, se iniciará una cuenta atrás que dará el comienzo de la partida.


*Figura 54. Esperando la cuenta atrás*

Antes de explorar las funcionalidades dentro de la partida, vamos a comentar primero cómo unirnos a una partida que otro usuario haya creado. Esta vez tenemos que pulsar en “Buscar Partidas”.


*Figura 55. Buscando partida*

Como podemos ver, hay una partida creada por otro usuario llamada PartidaDePrueba2, vamos a unirnos a ella pulsando en el botón.


*Figura 56. En la sala de partida creada por el otro usuario*

Ya estamos en la sala de la partida creada por el otro usuario, desde aquí es igual que en el caso anterior.

Ahora sí, vamos a pasar a explicar el funcionamiento dentro de la partida.

## Dentro de la partida

Y, para terminar, nos encontramos dentro de la partida. La parte donde más aspectos vamos a observar. Vamos a explicar brevemente en qué consiste el juego. El objetivo del juego es acabar con la vida de los oponentes. ¿Cómo? Con nuestras *Skillshots*. Podemos lanzarlas con ayuda de los joysticks que se encuentran a la derecha de nuestra pantalla. Si conseguimos que impacten contra un enemigo, le haremos daño. Tenemos que tener cuidado de que no nos dañen a nosotros, para ello, el jugador dispone de un joystick en la parte izquierda de la pantalla, con el cual puede mover al personaje por la arena. Aún hay más, cada cierto tiempo, en el suelo de la arena, aparecerán objetos aleatorios. Al recoger estos objetos, nuestro personaje experimentará cambios en sus estadísticas, que podemos ver en todo momento en nuestro HUD. Habrá objetos que aumenten nuestra velocidad y nuestro ataque, otros que lo hagan con la defensa. Pero no todos los objetos son perfectos, algunos tienen desventajas. Pueden bajar alguna estadística a la vez que suben otra. Estos efectos que tienen los objetos también les pertenecen a algunas *Skillshots*. Si te impacta alguna, puedes ser víctima, además del daño, de algún tipo de desventaja adicional.

Una vez vista la teoría, vamos a comenzar con la práctica. Primero, vamos a probar a mover a nuestro personaje con el joystick izquierdo.

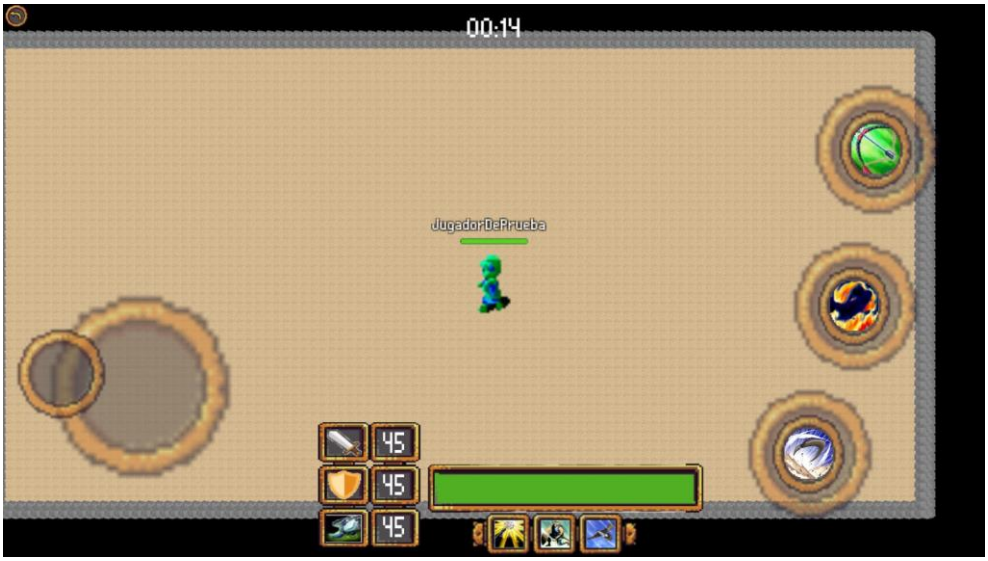


*Figura 57. Movimiento del personaje*

Ahora vamos a probar a lanzar todas las habilidades que tengamos asignadas.

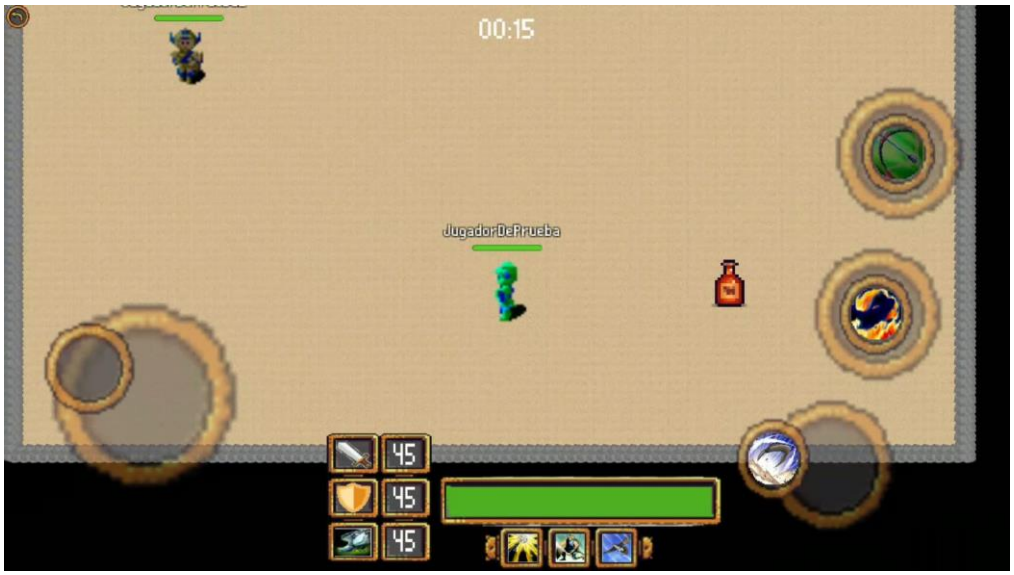


*Figura 58. Lanzando las habilidades*

Como podemos ver, la ilustración de los joysticks de las habilidades se apaga, y al cabo de un rato, se vuelven a iluminar. Esto se conoce como enfriamiento. Es un tiempo, durante el cual, no puedes volver a lanzar la habilidad. Las distintas habilidades tienen enfriamientos distintos, dependiendo de su grado de poder.


*Figura 59. JoystickSkillshot con enfriamientos*

Al cabo de un rato, si no lo ha hecho ya, aparecerá el primer objeto en el suelo de la arena, si nos movemos hasta él, desaparecerá y veremos afectadas nuestras estadísticas. Así como una pequeña imagen encima de nuestra barra de vida que indica por qué elementos estamos bajo su efecto en ese preciso instante.


*Figura 60. Durante la afección de un objeto*

Un tiempo después, el efecto desaparece, las estadísticas vuelven a la normalidad y la pequeña imagen de encima de la barra de vida también se va.


*Figura 61. Después de que desaparezca la afección del objeto*

Ya hemos explicado todas las mecánicas del juego, ahora solo falta enfrentarnos a otro usuario. Podemos aplicar todo lo aprendido. Al final de la partida, si tienes suerte, puede que veas el siguiente mensaje, significará que has ganado.

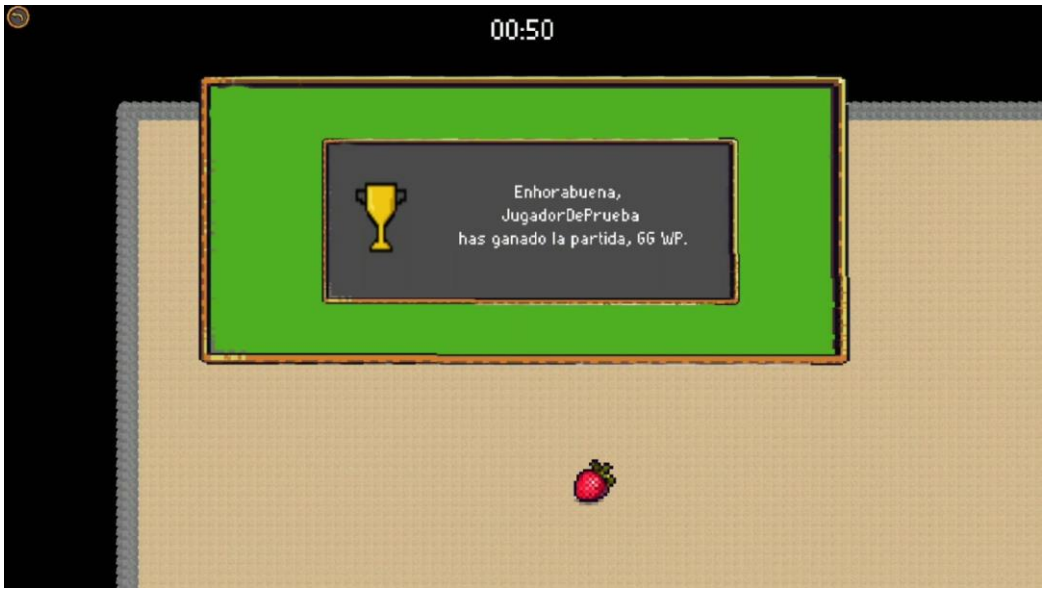


*Figura 62. Mensaje de victoria al final de la partida*