UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
CAMPUS D'ALCOI

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Politécnica Superior de Alcoy

Implementación y desarrollo de un videojuego de sigilo con infraestructura multijugador

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Vidal García, Eva

Tutor/a: Linares Pellicer, Jordi Joan

Cotutor/a: Izquierdo Doménech, Juan Jesús

CURSO ACADÉMICO: 2021/2022

Resum

Aquest projecte consisteix en el desenvolupament d'un videojoc en dues dimensions amb diverses funcionalitats multijugador cooperatiu entre dues persones i centrat en el sigil, dut a terme juntament amb 2 companys més. Aquest TFG se centra principalment a resoldre les complexitats que sorgeixen amb el model de jocs multijugador, els models de xarxa, els jugadors totals, el rang de latència i el disseny d'una bona escala de simulació sincronitzada.

Paraules clau: Unity, videojoc, sigil, cooperatiu, 2D, multijugador, servidor, xarxes, Netcode

Resumen

Este proyecto consiste en el desarrollo de un videojuego en dos dimensiones con varias funcionalidades multijugador cooperativo entre dos personas y centrado en el sigilo, llevado a cabo junto con otros 2 compañeros. Este TFG se centra principalmente en resolver las complejidades que surgen con el modelo de juegos multijugador, los modelos de red, jugadores totales, rango de latencia y diseño de una buena escala de simulación sincronizada.

Palabras clave: Unity, videojuego, sigilo, cooperativo, 2D, multijugador, servidor, redes, Netcode

Abstract

This project consists of the development of a two-dimensional cooperative multiplayer video game with various functionalities between two people and focused on stealth, carried out together with 2 other partners. This TFG focuses mainly on solving the complexities that arise with the multiplayer game model, network models, total players, latency range and design of a good synchronized simulation scale.

Key words: Unity, videogame, stealth, cooperative, 2D, multiplayer, server, networks, Netcode

Índice general

Índice general v
Índice de figuras vii
Índice de tablas vii

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 1
1.3 Metodología . . . 2
1.4 Estructura de la memoria . . . 3
1.5 Colaboraciones . . . 3

2 Estado del arte 5
2.1 Los inicios del videojuego . . . 5
2.2 Relevancia de los videojuegos en el momento actual . . . 5
2.3 Videojuegos de sigilo . . . 6
2.4 Videojuegos en línea . . . 6
2.5 Motores de videojuegos . . . 8

3 Análisis del problema 11
3.1 Explicación del videojuego . . . 11
3.2 Casos de uso . . . 12
3.3 Especificación de requisitos . . . 15
3.4 Plan de trabajo . . . 18

4 Diseño de la solución 21
4.1 Motor utilizado: Unity . . . 21
4.2 Topologías de red . . . 22
4.3 Sincronización del movimiento . . . 24
4.4 Sincronización de las animaciones . . . 24
4.5 Enemigos . . . 24
4.6 Objetos . . . 24
4.7 Cambio de escenas . . . 25
4.8 Cámara . . . 25
4.9 Interfaz . . . 25

5 Desarrollo de la solución 27
5.1 Topología de red . . . 27
5.2 Interfaz . . . 29
5.3 Movimiento . . . 30
5.4 Animaciones . . . 31
5.5 Enemigos . . . 31
5.6 Cambio de escenas . . . 32
5.7 Cámara . . . 33
5.8 Interacción con el escenario . . . 34

6 Pruebas 35
6.1 Pruebas Alpha . . . 35

6.2 Pruebas Beta 35

7 Conclusiones **37**

8 Relación con los estudios **39**

9 Trabajos futuros **41**

10 Glosario **43**

Bibliografía **45**

Apéndices

A Controles **47**

B Código relevante **49**

Índice de figuras

2.1 Shoplifting Boy (1979) y Metal Gear(1987) 7
2.2 Spasim (1973) 7
2.3 World of Warcraft (2004) 8
2.4 Interfaz de Unity 9
2.5 Interfaz Unreal Engine 10

3.1 Caso de uso Menú Principal 12
3.2 Caso de uso Menú de Pausa 13
3.3 Caso de uso de la protagonista 13
3.4 Caso de uso de los enemigos 14
3.5 Sprints en el desarrollo del proyecto 19

4.1 Topología Peer-to-Peer 22
4.2 Topología cliente host 23
4.3 Topología de servidor dedicado 23

5.1 Arquitectura usando Relay Server 27
5.2 Número de asignaciones al host en nuestro Relay Server 28
5.3 Interfaz Menú Principal y Crear Partida Online 29
5.4 Los 2 jugadores conectados 30
5.5 Llamada un cliente a un método ServidorRPC 31
5.6 Sprites de los enemigos 32
5.7 Escena del primer nivel 33
5.8 Cortina en el escenario 34

Índice de tablas

3.1 Requisito funcional 1 15
3.2 Requisito funcional 2 15
3.3 Requisito funcional 3 15
3.4 Requisito funcional 4 15
3.5 Requisito funcional 5 15
3.6 Requisito funcional 6 15
3.7 Requisito funcional 7 16
3.8 Requisito funcional 8 16
3.9 Requisito funcional 9 16
3.10 Requisito funcional 9 16
3.11 Requisito funcional 11 16

3.12 Requisito funcional 12 16
3.13 Requisito funcional 13 16
3.14 Requisito funcional 14 16
3.15 Requisito funcional 15 17
3.16 Requisito funcional 16 17
3.17 Requisito funcional 17 17
3.18 Requisito funcional 18 17
3.19 Requisito funcional 19 17
3.20 Requisito no funcional 1 17
3.21 Requisito no funcional 2 17
3.22 Requisito no funcional 3 17

A.1 Controles 47

CAPÍTULO 1
Introducción

El primer apartado de este Trabajo de Final de Grado consiste en la explicación de los motivos por los cuales se ha decidido realizar este proyecto, a la vez que se exponen los objetivos que se han querido cumplir y la metodología seguida durante el desarrollo del trabajo para lograrlos. Finalmente, se hace una explicación de la estructura que tiene el documento y de los otros estudiantes que han participado en este proyecto.

1.1 Motivación

Han sido varias las motivaciones que han llevado a la realización de este trabajo. Una de ellas es la afición personal por los videojuegos, siempre ha sido uno de mis hobbies principales jugar videojuegos y siempre he sentido curiosidad por como estaban hechos más a nivel técnico y en cuanto a programación y tecnologías utilizadas.

En la carrera también he cursado dos asignaturas relacionadas con los videojuegos que han hecho que despertara más mi interés para hacer un proyecto de esta índole y además con ellas he podido aprender mucho como se crean y desarrollan los juegos.

Finalmente, hablando con otros dos compañeros, llegamos a la conclusión de que se quería desarrollar este trabajo y que cada uno podía aplicar sus conocimientos para hacerlo más grande y complejo.

1.2 Objetivos

El principal objetivo de este TFG es el de elaborar un videojuego con cierta complejidad que pueda divertir y entretener al consumidor. Este objetivo primario consta y da forma a una serie de objetivos secundarios, que se describen a continuación:

- **Género de plataformas y sigilo:** El juego debe basarse en la temática de plataformas, que se caracterizan por tener que caminar, correr, saltar o trepar sobre una serie de plataformas. También tiene que estar inspirado en el sigilo. Este género se basa en esquivar enemigos sin ser detectado.

- **Videojuego multijugador:** El juego ofrecerá la posibilidad de jugar en modo cooperativo en línea con otros jugadores, de manera que se podrá disfrutar solo o con amigos.

- **Alta interactividad del entorno:** En la creación del juego se ha priorizado que los jugadores se sientan parte del entorno en el que se encuentran. Para conseguir esto

se permite que se puedan relacionar con varios objetos del mundo, no solo con los enemigos.

- **Enemigos reactivos a estímulos:** Los enemigos se programarán de tal forma que puedan reaccionar a las acciones del jugador, por lo que los enemigos con el mismo código podrán actuar en cualquier escenario y situación.

1.3 Metodología

El desarrollo de este videojuego se ha basado en una metodología ágil, que se determina por un desarrollo progresivo, en la cual se realizan diversas iteraciones, agregando y refinando diferentes componentes en cada una de ellas. Específicamente, nos hemos decantado por el modelo *Scrum*, que presenta más sprints con una menor carga de trabajo pero más pequeños y con mayor frecuencia. Podemos dividir esta metodología en las siguientes etapas:

- **Planificación:** Consiste en determinar el propósito por el cual se realiza cada sprint, o sea, los objetivos y desarrollo de UTs, riesgos, tiempos de entrega y pruebas de aceptación que se llevarán a cabo.
- **Desarrollo:** Durante esta etapa se ejecutan las UTs realizadas en el anterior paso y las pruebas de especificación se realizan, todo ello dentro del marco de tiempo dado para el sprint.
- **Revisión:** Cuando el sprint es completado, se solicita la retroalimentación del usuario. En esta etapa se muestra el progreso y se solicita una calificación que servirá para la siguiente fase.
- **Retroalimentación:** Se aplica al proyecto la información obtenida en la etapa anterior y se analizan las posibles modificaciones que se pueden llevar a cabo en la planificación de los siguientes sprints.

Esta metodología añade una gran velocidad y flexibilidad a los desarrollos de proyectos, especialmente en el caso de un trabajo colaborativo en el que participan más de una persona. La alta tasa de éxito, avalada por diversos estudios, es el motivo por el cual se elige adoptar esta forma de trabajar en el mundo laboral.

Cuando se quiere adaptar este tipo de modelo ágil para cualquier proyecto en una organización, se necesitan 3 figuras fundamentales:

- **Scrum Master:** Es el responsable de asegurar que el equipo cumpla con los objetivos proporcionados por el enfoque ágil al facilitar la relación entre el equipo de desarrollo y el cliente. Este individuo ha de tener una variedad de habilidades estratégicas y de comunicación, al actuar como el coach del equipo.
- **Product owner:** Esta figura se encarga de hacer todo lo posible por mantener la imagen del producto original, asegurando que se satisfagan las necesidades de sus consumidores y clientes. Su cometido es maximizar el valor de la entrega y es responsable de la interacción con el cliente y la gestión del *Backlog*.
- **Equipo de desarrollo:** Tienen una meta en conjunto, que es completar las tareas dentro del tiempo asignado, a la vez que una menor interacción con los clientes, únicamente cuando se requiera para comprender sus necesidades. Usualmente, los equipos se componen por un grupo de entre 5 y 9 empleados.

1.4 Estructura de la memoria

En esta sección se expondrán las diferentes partes que van a componer esta memoria, con la explicación correspondiente de cada una de ellas.

- El capítulo 1 incluye una **introducción** en la cual se explican las motivaciones para realizar este proyecto, así como la metodología utilizada a lo largo del proceso, la estructura que tendrá este documento y las colaboraciones de las que consta.
- El próximo capítulo consistirá en la presentación del **estado del arte**, en el cual se hablará de la historia que tienen los videojuegos, desde sus inicios hasta su relevancia y trascendencia actual, enfocándonos en los juegos que han tenido más repercusión en la historia, mencionando a su vez aquellos que son más importantes respecto a nuestro videojuego, por su parentesco en cuanto a género y temática.
- En el **análisis del problema**, se discutirá el juego en general y se pondrá especial énfasis en el inicio del proceso de desarrollo, la metodología de la cual se ha hecho uso y los requerimientos que se han considerado durante la vida del proyecto.
- El núcleo de este documento estará dividido en 2 partes: **diseño** y **desarrollo**, en las cuales se expondrá detalladamente cada elemento del juego. En la primera parte de estas secciones, las fases de diseño del juego y planificación se discutirán colectivamente, haciendo inca-pie en los elementos que se han considerado trascendentales para el desarrollo del juego y las soluciones implementadas. La fase de desarrollo, por otro lado, se centra en la implementación del videojuego en Unity, mostrando exactamente cómo se ha ido dando vida a los elementos que se comentan primeramente en la fase del diseño.
- Seguidamente, se presentará el capítulo de **pruebas**, que es la etapa final del desarrollo del proyecto, se expondrán las opiniones de los compañeros que probarán el juego y brindarán *feedback* sobre los elementos que encuentren menos atractivos o consideren que necesitan mejoras. Todas las críticas y las mejoras que se lleven a cabo a partir de ellas se expondrán en esta sección.
- A continuación, se presentan las **conclusiones** que se han alcanzado a lo largo del desarrollo del videojuego y la escritura de la memoria.
- Finalmente, se expondrá relación que ha tenido este proyecto con los estudios realizados y también se hablará de los posibles trabajos futuros a raíz de él, destacando las medidas que podrían agregarse si se dispusiera de mayor tiempo.

1.5 Colaboraciones

Este proyecto se ha llevado a cabo con otros dos alumnos, cada uno de los cuales ha sido responsable de crear diferentes componentes del juego hasta alcanzar el videojuego completo que se presenta. El trabajo realizado por cada uno de ellos ha sido el siguiente:

- **Eva Vidal García:** Implementación y desarrollo de un videojuego de sigilo con infraestructura multijugador. Sistema online para que el juego sea posible jugarlo con otro amigo.
- **Daniel De Castro Isasi:** Inteligencia artificial de los enemigos y jefe final, movimiento y comportamiento del protagonista, combate, iluminación y gestión de las animaciones.

- **Marcos Calero Domínguez:** Planificación, desarrollo y mantenimiento de un videojuego de sigilo Encargado principal de las interfaces e inventario, persistencia, interacción con el mapa, cuadros de texto y sonido.

CAPÍTULO 2
Estado del arte

Este capítulo tratará sobre la historia de los videojuegos, desde sus inicios, hace algunas décadas, hasta hoy en día y los desarrollos y novedades que han ido surgiendo a lo largo de los años. Posteriormente, comentaremos específicamente en los géneros a los que pertenece el juego que se ha desarrollado.

2.1 Los inicios del videojuego

La industria de los videojuegos es actualmente la industria del entretenimiento con mayores ingresos, genera más beneficios incluso que la industria cinematográfica y deportiva combinadas en Estados Unidos el año 2020 [1]. A pesar de que este número ha aumentado debido a la repercusión causada por la pandemia en los últimos años, los videojuegos no siempre tuvieron la repercusión e importancia que tienen en la actualidad.

Los orígenes de los videojuegos se remontan a la década de 1950, cuando se presentó el primer videojuego en la Feria Mundial de Nueva York en 1940, nombrado Nim. Casi 50.000 jugadores lo jugaron en seis meses que tuvo de vida; sin embargo, la primera consola no se comercializó hasta 1972. Esta consola tuvo un prototipo que se conoció como *The Brown Box*, acabó comercializándose con el nombre de *Magnavox Odyssey* en los Estados Unidos, fue sacada al mercado por Philips y fue un éxito comercial con 300.000 consolas vendidas, aunque solo duró 3 años en venta. En España se conoció como Overkai y llegó solo dos años después, en 1974, con 28 juegos en total.

Uno de los mayores contribuyentes al surgimiento de los videojuegos fue *Atari*, una empresa que se fundó en 1972. Atari fue la pionera en el campo de los juegos de género *arcade* y no fue hasta mediados de los 80 que empezó a perder su relevancia cuando surgieron 15 nuevas empresas en el sector.

Atari también fue quien promovió los videojuegos como pasatiempo en el ámbito doméstico al publicar la consola Atari VCS. Se lanzó al mercado con 10 sencillos juegos y con una ranura para cartuchos ROM, lo cual fue explotado por muchos desarrolladores a nivel mundial. Las ventas no fueron del todo satisfactorias para la empresa durante los 3 primeros años, hasta que se lanzó *Space Invaders* [2], un juego que revivió e impulsó las ventas de consolas a nuevas fronteras.

2.2 Relevancia de los videojuegos en el momento actual

Después de haber tratado la procedencia de la industria de los videojuegos y las compañías que los crearon y propulsaron, vamos a hacer énfasis en la importancia de estos

en los años más recientes, sobre todo en la repercusión que tuvieron en la época de confinamiento y pandemia causada por el COVID-19.

La situación excepcional que se dio en los primeros mese de pandemia, cuando a las personas no se les permitía salir de casa, provocó que la mayoría de la gente considerara necesario buscar otros medios de contacto y comunicación con los demás. Durante este tiempo es cuando los videojuegos cobraron mucha más importancia hasta hoy en día, ya que la mayoría de la gente recurre a ellos para resolver el obstáculo de la distancia física con otras personas [3], normalmente a través de puros juegos multijugador que se pueden jugar con amigos.

Asimismo, según muchos estudios, libros [4] y artículos publicados, sobre todo después del confinamiento por la pandemia, se ha demostrado que las destrezas de competencia y socialización que brindan los videojuegos promueven el bienestar de la gente que se dedica a jugarlos, interviniendo de forma muy positiva sobre su salud mental.

2.3 Videojuegos de sigilo

Un videojuego de sigilo es un juego en el que se anima al participante a agredir a los adversarios sin que este los perciba o a sortearlos totalmente. En este tipo de videojuegos, por norma general, suele estar la posibilidad de esconderse, agacharse, disfrazarse, entre otros, para impedir ser expuesto por los enemigos. También, en contraposición con los típicos juegos de acción, dan al jugador muy pocas o ninguna posibilidad de enfrentamiento cuerpo a cuerpo contra los enemigos, incitándole a buscar otras alternativas para solucionar las disputas utilizando las herramientas que tiene a su disposición en cada momento. Los juegos de este género enfatizan un diseño de niveles óptimo, que los enemigos dispongan de una inteligencia artificial más reactiva y una gran diversidad de herramientas que se le tienen que proporcionar al jugador.

Shoplifting Boy, lanzado en 1979, fue el primer videojuego considerado de género sigilo. Este juego trataba de robar la mayor cantidad de elementos posibles de una tienda, evitando las regiones de visión de los enemigos y ser arrestado por la policía. En 1981 se lanzó *Castle Wolfenstein*, en el cual el protagonista tiene que viajar a través de un castillo repleto de guardias para acceder a los planes secretos del enemigo y escapar sin ser detectado. Además, otro de los juegos iniciales y más relevante de este género fue el primer videojuego de la saga *Metal Gear*, sacado al mercado por la empresa Konami y creado por Hideo Kojima, el cual sigue siendo una de las figuras más importantes de la industria en la actualidad. Se puede decir que fue el primer juego en ofrecer este tipo de sigilo a un público más amplio en el que dominaba los juegos de acción.

El género ha ido creciendo y cobrando relevancia a lo largo de los años, y es uno de los más influyentes en la actualidad, que cuenta con juegos como la saga *Splinter Cell*, que dispone de 7 entregas comercializadas por Ubisoft. Una de las aventuras modernas más exitosas, tanto en cuanto a ventas como críticamente, es *Dishonored*, con 2 volúmenes principales. Esta historia tiene un gran diseño de niveles y un amplio abanico de posibilidades que ofrece al jugador en cualquier situación.

2.4 Videojuegos en línea

Los videojuegos en línea son aquellos que permiten la interacción de dos o más jugadores al mismo tiempo a través de la conexión por medio de una red que puede ser Internet o una red de área local. Hasta la década de los años 70, los juegos multijugador se basaban en jugadores que jugaban en la misma pantalla, hasta que finalmente en 1973

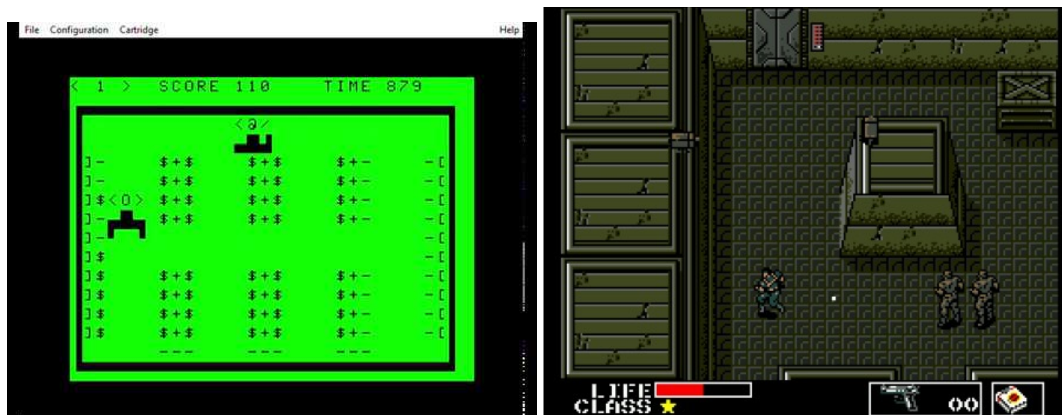

Figura 2.1: Shoplifting Boy (1979) y Metal Gear(1987)

se estrenó *Empire* [5], un juego estratégico de combate por turnos donde podían jugar incluso ocho jugadores, que se creó para el sistema de red PLATO.

En 1973, Jim Bowery lanzó *Spasim*, también para PLATO. Era un juego espacial de 32 jugadores, considerado el primer ejemplo de un juego multijugador en 3D; representa uno de los primeros pasos en el camino tecnológico hacia Internet y los juegos multijugador en línea tal y como los conocemos hoy.

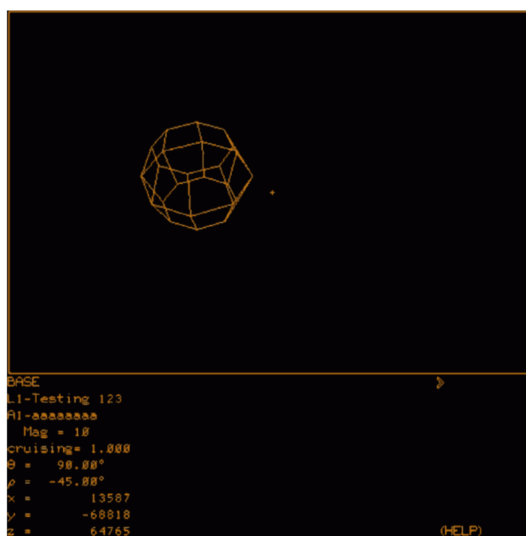

Figura 2.2: Spasim (1973)

En 1979 un grupo de alumnos universitarios crearon una versión informática del juego de rol *Dungeons and Dragons*, utilizando un ordenador para jugar en redes telemáticas, creándose así un tipo nuevo de juegos nombrados MUD (Multi-User Dungeons o Domains) que se desarrolló rápidamente por el aún poco conocido Internet, surgiendo así las primeras comunidades virtuales.

No fue hasta los años 90, cuando Internet empezó a tener mayor disponibilidad al público, que la expansión de los juegos en línea pudo comenzar; con títulos famosos como *Nexus: The Kingdom of the Winds* (1996), *Quakeworld* (1996), textitUltima Online (1997), *Starcraft* (1998), *Counter-Strike* (1999) y *EverQuest* (1999).

A partir de los años 2000 el coste de los servidores e Internet se redujo notablemente, lo que provocó que los juegos online se siguieran desarrollando rápidamente y se creara el género MMO (Massive Multiplayer Online Games). *World of Warcraft* (2004) dominó gran parte de la década y superó los 10 millones de suscriptores. Varios MMOs intentaron

seguir los pasos de *World of Warcraft*, como *Star Wars Galaxies*, *Warhammer Online*, *Guild Wars 2* y *Star Wars: The Old Republic*, pero no obtuvieron el mismo impacto en el mercado.

Figura 2.3: World of Warcraft (2004)

Surgió en esta década también un nuevo tipo de juegos en línea. *Defense of the Ancients* (2003) introdujo los juegos MOBA (Multiplayer Online Battle Arena). *DotA* fue un mod (abreviatura de "modification") creado por la comunidad de jugadores y basado en *Warcraft III* ganó popularidad, pero como estaba ligado a la propiedad de *Warcraft*, otros creadores empezaron a desarrollar sus propios MOBAs y así nacieron League of Legends (2010) y Dota 2 (2013).

A lo largo de la última mitad de 2010, Hero Shooter, una variante de los juegos shooter creada a raíz de los MOBA y los shooters más antiguos, se hizo bastante popular con la publicación de Battleborn y Overwatch en 2016. El género siguió creciendo con juegos como Paladins (2018) y Valorant (2020).

Los juegos de estilo Battle Royale se hicieron también muy populares con la publicación de PlayerUnknown's Battlegrounds (2017), Fortnite Battle Royale (2017) y Apex Legends (2019), y así ha seguido hasta hoy en día.

2.5 Motores de videojuegos

Una de las partes más importantes para desarrollar un videojuego es la elección del motor del que se hará uso. Un motor para videojuegos se refiere a una colección de bibliotecas de programación que permiten el diseño, la creación y el rendimiento de un videojuego, con el objetivo de facilitar el proceso. Toda herramienta gráfica debe proporcionar funciones esenciales para el programador, ofreciendo funcionalidades como el motor para renderizado de tanto 2D como 3D, colisión de objetos y su respuesta, animación, la conexión con la red necesaria para juegos multijugador, entre otros. También son muy apropiados en el caso de que se requiera volver a usar o ajustar una parte de algún juego creado con anterioridad para trabajar en un proyecto nuevo, optimizando el desarrollo.

Seguidamente, se explican con más detenidamente algunos de los motores más populares dentro del mundo de los videojuegos, que se emplean para los diferentes tipos de juegos e importe, tanto los juegos con más renombre creados por un gran equipo de personas como los juegos independientes más pequeños.

- **Unity:** Actualmente, es uno de los motores más populares en la industria, para todos los videojuegos de diferentes géneros y estilo. Unity sale al mercado en 2005,

destinado únicamente para el desarrollo de videojuegos 3D, a pesar de que los desarrolladores ingeniaron formas de lograr un 2D gracias a técnicas únicas como agregar texturas a pantallas planas, finalmente en 2013, se integró soporte de forma oficial para desarrollar juegos 2D.

El éxito de este motor viene dado por tres principales elementos:

El primero de ellos es la facilidad de uso del programa, ya que cuenta con un gran número de tutoriales que informan sobre todas las funcionalidades de la herramienta y la ponen al alcance de todos los tipos de usuario.

Seguidamente, destaca la posibilidad de publicar los videojuegos en más de 25 plataformas, incluidas Windows, Xbox One, Nintendo Switch y Android entre muchas otras. Esto facilita que la mayoría de desarrolladores se decanten por este motor.

El último de los factores es, como se ha mencionado anteriormente, la flexibilidad que Unity proporciona a la hora de desarrollar cualquier tipo de juego, tanto 2D como 3D, en contraposición con otros motores que están más enfocados a un solo tipo de desarrollo, como Unreal Engine, el cual está más centrado para juegos 3D o RPGMaker, que sirve para desarrollar RPGs en 2D.

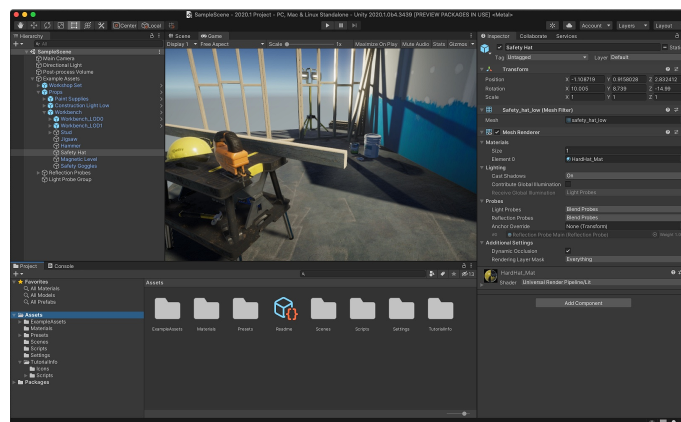

Figura 2.4: Interfaz de Unity

- **Unreal Engine:** Fue creado por Epic Games y sacado al mercado en 1998. Este, junto con Unity, es uno de los motores de videojuegos más populares. Unreal Engine tiene muchas funcionalidades dignas de comentar como su asombrosa precisión gráfica, una muy buena implementación de scripts visual denominada *blueprint*, un marco multijugador muy consistente y un gran repertorio de tutoriales y documentación oficial.

  En contraposición, una de las desventajas que tiene este motor es la dificultad de aprendizaje que presenta, debido, entre otros, a que su el lenguaje de programación que utiliza es C++.

- **Godot:** Es un motor de código abierto y completamente gratuito que ha sido creado completamente por la comunidad y actualizada de acuerdo a los requisitos generales de los desarrolladores que hacen uso de ella.

  Los lenguajes de programación admitidos oficialmente incluyen Visual Scripting, C# y C++, pero GDScript es el lenguaje oficial para el motor y del que recomiendan hacer uso en el sitio web oficial del motor.

- **Motores particulares:** Una gran cantidad de compañías, sobre todo las más importantes de la industria y con más beneficios, desarrollan sus propios videojuegos basándose en un motor propio que actualizan cada cierto tiempo y les permiten

Figura 2.5: Interfaz Unreal Engine

mantener el control total sobre el motor en todo momento. Alguno de los motivos para preferir un motor particular puede ser la reducción de costos de producción, debido a que la misma herramienta o sus variantes pueden ser utilizadas para diferentes juegos en una misma empresa y, por tanto, ya no es necesario pagar los gastos de usar un motor externo.

CAPÍTULO 3
Análisis del problema

A lo largo de este capítulo se realizará una investigación para determinar las necesidades del proyecto que se va a llevar a cabo. Para conseguir esto es necesario que se elaboren los requisitos tanto funcionales como no funcionales que determinaran los requisitos del resultado final. Se va a empezar con una explicación de la intención del juego, posteriormente se analizarán los casos de uso y, haciendo uso de ellos, se trazarán los requerimientos que hacen falta. Finalmente, se elaborará un plan de trabajo y presupuesto de implementación del proyecto para tener una idea del costo de este proyecto.

3.1 Explicación del videojuego

El propósito primordial del videojuego que se presenta es controlar a la guerrera que desempeña el papel del personaje principal para conseguir superar un conjunto de diferentes niveles. Durante el proceso, la protagonista se topará con varios enemigos repartidos por los niveles, que harán todo lo necesario para acabar con ella y que no consiga completar su meta.

Se puede afirmar que este videojuego pertenece al género plataformas y sigilo en dos dimensiones, donde la cámara del juego se mueve siguiendo a la protagonista a lo largo del nivel y mostrando un área limitada de este, teniendo una vista en 2D y con desplazamiento lateral. Al igual que en otros juegos de esta índole, el usuario tendrá que escalar, saltar, esconderse y luchar para conseguir el objetivo de llegar al final de cada nivel. En conjunto el proyecto se compone de tres niveles, el primero es una experiencia de tipo tutorial y desde este punto de partida se va incrementando la dificultad y el tamaño del nivel hasta llegar al jefe final, que se encuentra en el área final del último nivel.

A su vez, como se explicó con anterioridad, el videojuego posee un importante elemento de sigilo. Es por eso que las protagonistas tendrán muchas herramientas en los niveles que se utilizarán para evitar enemigos tanto como sea posible. Algún ejemplo de estas herramientas son una bola de agua para extinguir la llama de las antorchas a señuelos para atraer a los enemigos que los vean. Aparte, el escenario será diseñado también teniendo esto en cuenta, lo que permitirá a los jugadores esconderse detrás de las cortinas para ocultar su posición.

3.2 Casos de uso

Con anterioridad al inicio del desarrollo del juego, se han llevado a cabo un conjunto de casos de uso que sirven para representar las funciones requeridas en el videojuego. Dichos casos de uso cubrirán las acciones que los jugadores serán capaces de poner en práctica desde el menú principal, menú de pausa, las acciones que efectúa la protagonista y las acciones que llevarán a cabo los enemigos. Seguidamente, se presentan estos casos de uso junto a una aclaración de las posibilidades que hay en ellos.

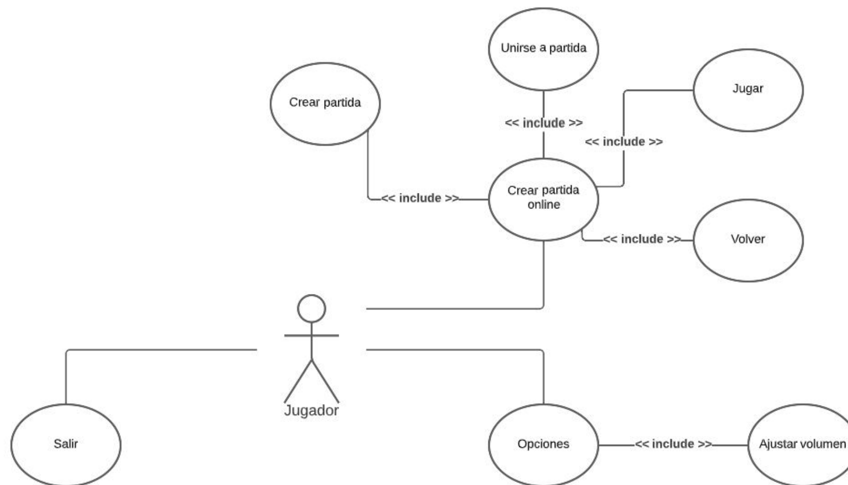

Figura 3.1: Caso de uso Menú Principal

- **Crear partida online:** Al presionar este botón, se abrirá un nuevo menú con nuevas opciones para iniciar una partida o conectarse a ella. Habrá también otro botón para volver de nuevo al menú principal
- **Crear partida:** Se crea una partida multijugador nueva de la cual el usuario que presione el botón será el host.
- **Unirse a partida:** Este botón sirve para que un cliente se conecte que a una partida que ya ha empezado un host.
- **Jugar:** Una vez los 2 usuarios están conectados, este botón hace que se empiece la nueva partida y los 2 aparezcan en el inicio del juego.
- **Volver:** Ejecuta la acción de volver al Menú Principal.
- **Opciones:** Se entra en el apartado de Opciones. Desde aquí existe la posibilidad de subir o bajar el volumen general del juego a través de la opción **Ajustar volumen**.
- **Salir:** Causa que el usuario se desconecte del juego.

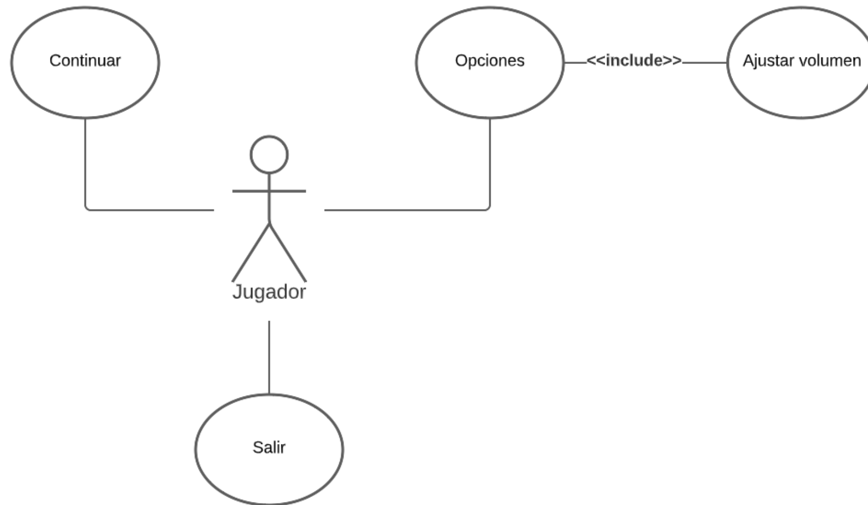

Figura 3.2: Caso de uso Menú de Pausa

- **Continuar:** Con este botón el usuario cierra el menú de pausa y reanuda el juego en el mismo instante donde lo había pausado.
- **Opciones:** El jugador entra en el menú de opciones desde donde si lo requiere puede ajustar el volumen que tiene el juego.
- **Salir:** El jugador cierra el juego y vuelve al Menú Principal.

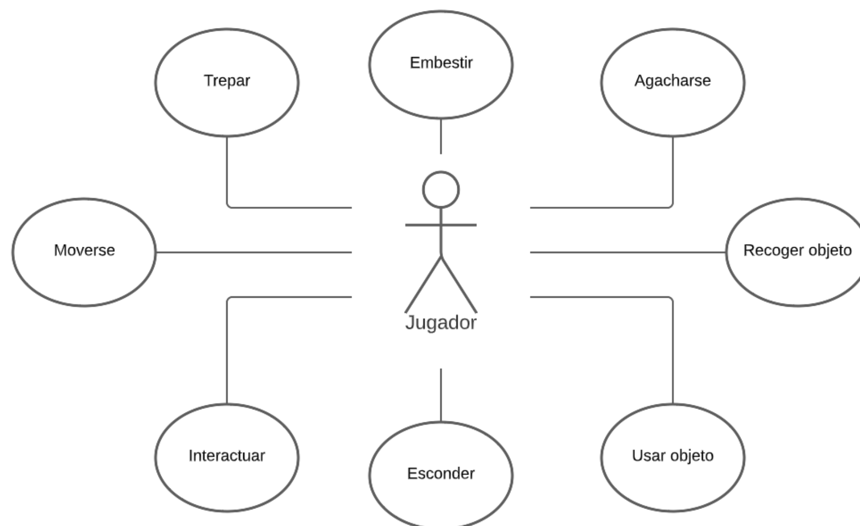

Figura 3.3: Caso de uso de la protagonista

- **Moverse:** La protagonista tiene que ser capaz de desplazarse por el nivel.
- **Embestir:** Se puede realizar un movimiento rápido hacia delante o hacia atrás que es capaz de derribar a los enemigos que haya en ese recorrido.
- **Trepar:** El jugador es capaz de subir y bajar por las escaleras que hay a lo largo de los niveles.

- **Agacharse:** La protagonista puede agazaparse para que sea más difícil ser detectada por los enemigos; sin embargo, esto provocará que se desplace más lentamente.
- **Esconderse:** El jugador puede usar las cortinas que hay en los niveles para esconderse detrás de ellas y no ser detectado.
- **Recoger objeto:** Los objetos que se encuentra el jugador a lo largo de los niveles aparecen automáticamente en su inventario al pasar sobre ellos.
- **Usar objeto:** Los objetos que se han recogido pueden ser usados seleccionándolos a través del inventario.
- **Interactuar:** La protagonista puede ejercer acciones sobre objetos situados en los diversos niveles.

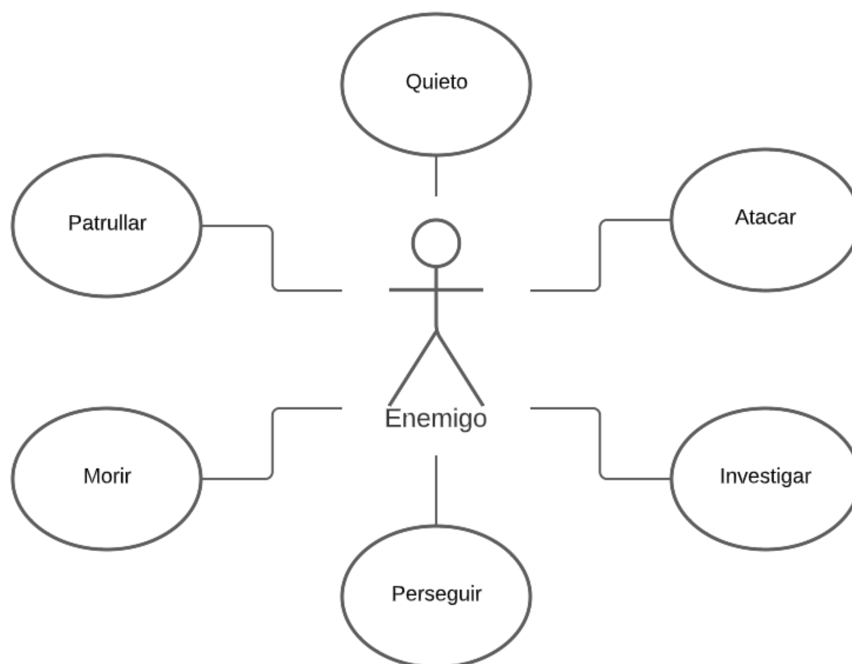

Figura 3.4: Caso de uso de los enemigos

- **Patrullar:** El enemigo se desplaza por el nivel cambiando el sentido de la marcha cuando se topa con algo.
- **Quieto:** El enemigo no se mueve.
- **Atacar:** El adversario realiza el ataque que sea correspondiente restando vida a la protagonista.
- **Investigar:** Se produce cuando se aproxima a la ubicación de otro objeto para comprobar qué sucede.
- **Perseguir:** El enemigo se camina sin cesar hacia donde esté la protagonista.
- **Morir:** Cuando la vida del enemigo está a 0 desaparece del mapa.

3.3 Especificación de requisitos

Esta sección describe los requisitos con los que cuenta el proyecto. Estos requisitos se componen por un identificador, un pequeño resumen del mismo y su descripción.

Requisito funcional 1	
Nombre	Crear partida online
Descripción	El sistema deberá abrir otro menú desde el cual se permitirá crear una partida online y poder jugar al juego

Tabla 3.1: Requisito funcional 1

Requisito funcional 2	
Nombre	Crear partida
Descripción	El sistema tendrá que crear una sesión nueva, la cual *hosteará* uno de los usuarios

Tabla 3.2: Requisito funcional 2

Requisito funcional 3	
Nombre	Unirse a partida
Descripción	La aplicación tiene que ofrecer la opción de que un cliente pueda unirse a una sesión iniciada por el host

Tabla 3.3: Requisito funcional 3

Requisito funcional 4	
Nombre	Jugar
Descripción	El sistema deberá permitir que una vez se haya iniciado la sesión se empiece la partida y se pueda jugar al juego

Tabla 3.4: Requisito funcional 4

Requisito funcional 5	
Nombre	Volver al menú
Descripción	El sistema tiene que permitir al usuario volver al menú principal desde el menú de pausa, al morir o al acabar un nivel

Tabla 3.5: Requisito funcional 5

Requisito funcional 6	
Nombre	Salir del juego
Descripción	La aplicación ha de permitir que los jugadores puedan desconectarse del juego a través del menú principal

Tabla 3.6: Requisito funcional 6

Requisito funcional 7	
Nombre	Menú de pausa
Descripción	La aplicación debe ofrecer la posibilidad de entrar al menú de pausa en cualquier momento dentro de los niveles, deteniendo así el juego de manera temporal

Tabla 3.7: Requisito funcional 7

Requisito funcional 8	
Nombre	Morir
Descripción	Cuando la vida de alguno de los 2 jugadores se reduce a 0, el sistema tiene que mostrar el menú de muerte para ambos clientes.

Tabla 3.8: Requisito funcional 8

Requisito funcional 9	
Nombre	Recoger objeto
Descripción	Los usuarios deberán poder añadir objetos a su inventario al acercarse a ellos cuando están en el suelo del nivel

Tabla 3.9: Requisito funcional 9

Requisito funcional 10	
Nombre	Usar objeto
Descripción	Los jugadores podrá utilizar un objeto siempre y cuando se localice en su inventario y esté previamente seleccionado.

Tabla 3.10: Requisito funcional 9

Requisito funcional 11	
Nombre	Atacar
Descripción	Los usuarios serán capaces de atacar a los enemigos mientras se tenga el objeto 'Espada' equipado desde el inventario

Tabla 3.11: Requisito funcional 11

Requisito funcional 12	
Nombre	Embestida
Descripción	El jugador podrá realizar una carga contra los enemigos, pero no se permite utilizar repetidamente, se debe esperar unos segundos

Tabla 3.12: Requisito funcional 12

Requisito funcional 13	
Nombre	Abrir cofre
Descripción	Si los usuarios se encuentran ubicados cerca de un cofre, deberán poder abrirlo

Tabla 3.13: Requisito funcional 13

Requisito funcional 14	
Nombre	Interactuar con palancas
Descripción	Si los jugadores se encuentran en una posición cercana a una palanca, podrán tirar de ella para activarla.

Tabla 3.14: Requisito funcional 14

Requisito funcional 15	
Nombre	Siguiente nivel
Descripción	El juego debe permitir acceder al nivel siguiente una vez se haya llegado al final de aquel en el que el jugador se encuentre

Tabla 3.15: Requisito funcional 15

Requisito funcional 16	
Nombre	Guardar progreso
Descripción	Al completar el nivel se deberá guardar la información del personaje, de forma que se mantengan los objetos que este tenga en el inventario de cara a la carga del siguiente nivel. Estos datos guardados deben mantenerse en caso de cerrar el juego para poder continuar la partida

Tabla 3.16: Requisito funcional 16

Requisito funcional 17	
Nombre	Sonidos
Descripción	Deberán existir sonidos representativos para las acciones principales del juego, los cuales se reproducirán en los momentos oportunos

Tabla 3.17: Requisito funcional 17

Requisito funcional 18	
Nombre	Movimiento
Descripción	Tanto el jugador como los enemigos deben ser capaces de desplazarse por el escenario

Tabla 3.18: Requisito funcional 18

Requisito funcional 19	
Nombre	Mostrar rango de lanzamiento
Descripción	En el caso de objetos concretos, se deberá representar gráficamente la trayectoria del lanzamiento previamente a usar el objeto

Tabla 3.19: Requisito funcional 19

Requisito no funcional 1	
Nombre	Corrección funcional
Descripción	El videojuego deberá ser capaz de funcionar en cualquier ordenador

Tabla 3.20: Requisito no funcional 1

Requisito no funcional 2	
Nombre	Accesibilidad
Descripción	Aquellas personas menos experimentadas en los videojuegos deberían ser capaces de disfrutar con él, por lo que las mecánicas del juego deberán ser sencillas y fácilmente entendibles

Tabla 3.21: Requisito no funcional 2

Requisito no funcional 3	
Nombre	Disponibilidad
Descripción	La aplicación se encontrará en funcionamiento mientras el jugador no cierre su ejecución

Tabla 3.22: Requisito no funcional 3

3.4 Plan de trabajo

Esta sección final, previa a explicar el diseño del juego, habla sobre el método ágil, más concretamente sobre el sistema *scrum*, para el desarrollo del videojuego que se presenta. Se ha distribuido este proceso en varios *sprints* principales y se les ha concedido un mes de tiempo para completar cada uno de ellos. Además, se ha añadido otro *sprint* que se centra en la producción previa al juego, llamado *Sprint 0*. Este incluye la lluvia de ideas llevada a cabo, el desarrollo de cada una de las tareas que se debían realizar y la creación de un primer diagrama de Gantt con una valoración de los tiempos necesarios para completar las metas establecidas en el mismo *sprint*.

Finalmente, antes de explicar el trabajo realizado en cada iteración, es interesante señalar que se ha hecho uso de una técnica de prototipos durante todo el proceso de desarrollo. Es decir, en lugar de diseñar un nivel y construir cada mecanismo según sea necesario, se parte de un desarrollo muy elemental, en el que cada uno de estos mecanismos se implementa de forma individual, combinándolos todos juntos cuando se logra el rendimiento deseado de cada uno de ellos.

Aquí se exponen cada uno de los *sprints* que se han llevado a cabo durante la realización del proyecto.

- **Sprint 0:** Este *sprint* se ha llevado a cabo antes de empezar a desarrollar el juego y en él se realizan una serie de tareas relevantes. La tarea inicial supuso una lluvia de ideas entre los diferentes miembros del equipo en la que se trataron las ideas como la temática del videojuego y los mecanismos que deberá tener. Teniendo esto en cuenta, se permite planear el proyecto de forma anticipada, previniendo que puedan surgir problemas relacionados con esto en un futuro.

  A continuación, se deberá separar estas materias y mecanismos en diferentes subtareas para separar el trabajo que se tiene que llevar a cabo y planificar de forma correcta. Finalmente, se generará un diagrama de Gantt estableciendo el tiempo estimado para cada una de estas tareas y organizar de forma óptima todo el proyecto.

- **Sprint 1:** En este *sprint* comienza verdaderamente la creación del juego. Como se ha mencionado anteriormente, este desarrollo se lleva a cabo a través de la implementación de prototipos, de forma que cada mecánica se pueda probar de manera individual antes de implementarla en el juego y en los propios niveles. Con el objetivo de lograr esto, durante este *sprint* se ha diseñado una escena base que ha permitido verificar el óptimo desempeño de dichas funcionalidades.

  A lo largo de este *sprint* nos hemos centrado en la función de recabar información sobre como hacer funcionar un sistema multijugador en un juego desarrollado con Unity. Una vez obtenida la información necesaria, ahora sí, se puede pasar a desarrollar el videojuego en sí. El primer paso para ello es decidir la topología de red que se quiere implementar en el proyecto, instalar los paquetes necesarios en Unity para que una conexión multijugador sea posible y crear la interfaz necesaria junto con los *scripts* de código que hagan posible la conexión de otro jugador.

- **Sprint 2:** Este *sprint* es el más sustancioso de todos, puesto que dentro de este lo primero que se ha llevado a cabo ha sido la sincronización del movimiento entre los 2 jugadores, esto significa que, cuando un jugador se mueve, el otro puede ver en su pantalla como esto ocurre. Cuando esto se ha logrado, el siguiente paso ha sido el de sincronizar todas las animaciones que realiza la protagonista, cuando salta, corre, ataca, entre otros; ya que las animaciones funcionan de forma diferente al movimiento.

- **Sprint 3:** En esta fase del juego se ha llevado a cabo la sincronización de los enemigos para los 2 jugadores, para que aparezcan los mismos enemigos a ambos usuarios y el estado que tengan sea el mismo, también se modificó su comportamiento para que siguieran y atacaran al jugador que encontraran más cerca. También se lleva a cabo modificaciones en algunos elementos de los escenarios, como palancas para que asimismo estuvieran sincronizados entre los jugadores. En este *sprint* también se ha realizado un cambio en la topología de la red y se sincronizó la vida de los jugadores en la interfaz. Finalmente, se aplican todos los cambios realizados en el diseño del juego a los niveles que se habían implementado, permitiendo que se puedan jugar de forma multijugador.

- **Sprint 4:** Esta última etapa, personas externas al proyecto se dedicarán a probar el juego y brindarán información sobre varios errores, así como mejoras para aplicar al juego final.

Para distribuir los *sprints* de manera clara se ha hecho uso de Trello, que es una herramienta en línea que se utiliza para gestionar proyectos colocando fichas en distintas listas con la intención de formar un tablero. Cada tarjeta interpreta una distinta labor para completar, y se pueden mover fácilmente por el listado. En cuanto a la ordenación de este proyecto, se ha distribuido para que todas las tareas comiencen en la lista de *backlog*. Desde aquí, las tarjetas que coincidan con el *sprints* presente se colocarán en la nueva lista, donde solo se moverán una vez concluidas o, por el contrario, se rechazarán por diversas razones.

Aparte, todas ellas, además de moverse de la lista donde se encuentran, es posible añadir una descripción, un comentario, una fecha de finalización y una etiqueta que la especifique, entre otras opciones. En nuestro caso, con este trabajo, se ha utilizado la colorimetría de las tarjetas de la siguiente manera:

- **Rojo:** Ocurrió un error en las pruebas que debe ser solucionado si se desea seguir con la tarea.

- **Naranja:** Se sigue trabajando en la labor.

- **Amarillo:** Trabajo terminado, pero se ha requerido realizar algunas modificaciones debido a las comprobaciones llevadas a cabo.

- **Verde:** La tarea se ha completado y se han superado todas las pruebas programadas.

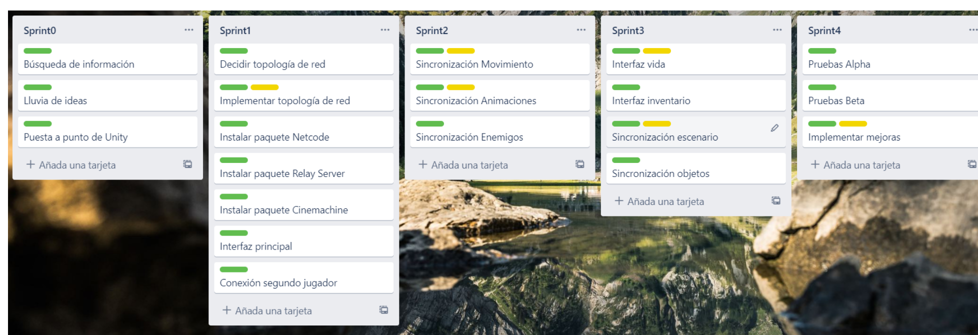

Figura 3.5: Sprints en el desarrollo del proyecto

CAPÍTULO 4
Diseño de la solución

Este capítulo se compone por la explicación de todos los componentes incluidos, junto con las interacciones entre cada uno de ellos. Como un proyecto conjunto, aunque todos los elementos del juego se explican resumidamente, nos enfocaremos concretamente en aquellos propios de este TFG.

4.1 Motor utilizado: Unity

Previamente, se discutió la importancia de elegir un motor de videojuegos cuando se desarrollan juegos y se revelaron algunas de las herramientas más populares y utilizadas en la industria. En el momento de decidir qué herramienta usar para este proyecto, hay varias razones a considerar, entre las cuales está por ejemplo el tipo de juego que se quiere conseguir y el nivel de experiencia de los integrantes del equipo. Teniendo todo esto en cuanta, finalmente se escogió Unity, una tecnología de la cual ya se había hecho uso en otras asignaturas y en proyectos personales.

Además, Unity ahora cuenta con una serie de características que apoyan el desarrollo de videojuegos 2D del estilo que se ha elegido, entre las cuales se encuentra la capacidad de hacer uso de *Tilemaps* para diseñar el nivel, los algoritmos de inteligencia artificial que se han hecho uso para implementar el comportamiento de los enemigos, o las herramientas y facilidades que incluye Unity o facilitadas por otros desarrolladores para implementar una topología multijugador.

Por otro lado, la meta más importante de este proyecto es lograr hacer un juego multijugador en el que puedan jugar a la vez 2 personas y, para lograr esto, Unity dispone de los paquetes, servicios e infraestructuras necesarias para conseguirlo. Como por ejemplo el servicio Relay [9] de Unity, que sirve para conectar a los jugadores sin la necesidad de disponer de un servidor dedicado, lo cual dispararía los costes del proyecto. Este servicio permite comunicaciones UDP de escucha directa P2P fáciles y seguras entre ambos jugadores.

Unity también dispone de Netcode [8], una biblioteca de redes que se ha usado a lo largo de todo el proyecto, creada por y para el motor de juegos de Unity, con la finalidad de abstraer las redes. Permitiendo a los desarrolladores concentrarse en su juego en lugar de protocolos de bajo nivel y marcos de trabajo en red.

Asimismo, como Unity tiene mucha popularidad dentro del ámbito *indie* y como motor para iniciarse en el desarrollo de videojuegos, es posible conseguir una amplia información de gran utilidad sobre las áreas de conocimiento en las cuales se ha necesitado ayuda, haciendo uso vídeos donde otros desarrolladores hacen tutoriales basados en su experiencia, o a través de varios foros o sitios de documentación oficial [7]. Además, exis-

ten soluciones a muchas preguntas o problemas con los que otros desarrolladores ya se han topado previamente en páginas como *Stack Overflow*.

Finalmente, otra de las razones por la que se ha escogido Unity es que su licencia es gratuita mientras no se obtenga más de 100.000 $ de ingresos al año. Asimismo, cuenta con un gran soporte que brinda diversos recursos y herramientas de la tienda oficial de recursos de Unity, llamada *Unity Asset Store* [10], muchos de los cuales son gratuitos.

4.2 Topologías de red

La topología de la red define la forma en que se organiza una red, incluyendo la forma en la que los enlaces y los nodos se relacionan entre ellos, la vez que los clientes y los hosts, y las máquinas físicas y virtuales. Las topologías de red posibles para crear un juego multijugador son las siguientes:

- **Multijugador local:** Consisten en juegos multijugador locales que se pueden jugar en la misma pantalla, como un televisor, usando solo el *runtime* del cliente.
- **Peer-to-Peer (P2P):** Una red peer-to-peer se crea cuando dos o más PCs están conectados y comparten recursos sin pasar por una computadora servidor.

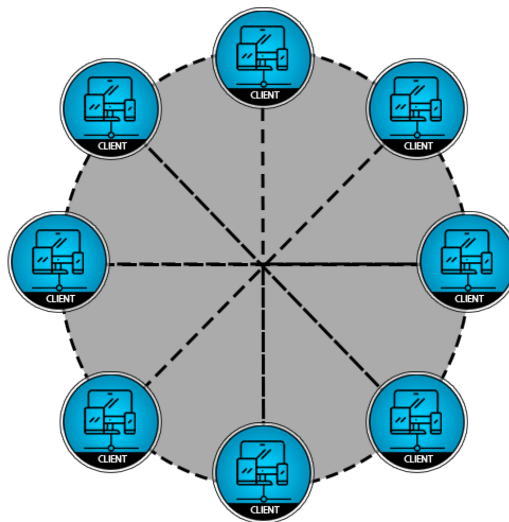

Figura 4.1: Topología Peer-to-Peer

- **Un cliente es el host (servidor de escucha):** El servidor de escucha se ejecuta en el mismo proceso que un cliente del juego.

  Un *host* es similar a un servidor, donde el servidor también es un jugador y .ªnfitrión"del juego. Esta topología se usa comúnmente en una *LAN* donde las personas se conectan a la misma red y un jugador aloja el juego (por ejemplo, Counter Strike).

  Funcionan como servidores dedicados, pero normalmente tienen la desventaja de tener que comunicarse con jugadores remotos a través de la conexión a Internet interna del jugador anfitrión. El rendimiento también se reduce por el simple hecho de que la máquina que ejecuta el servidor también genera una imagen de salida del juego.
- **Servidor dedicado:** Los servidores dedicados simulan el juego sin admitir entrada o salida directa, excepto la necesaria para su administración. Los jugadores deben

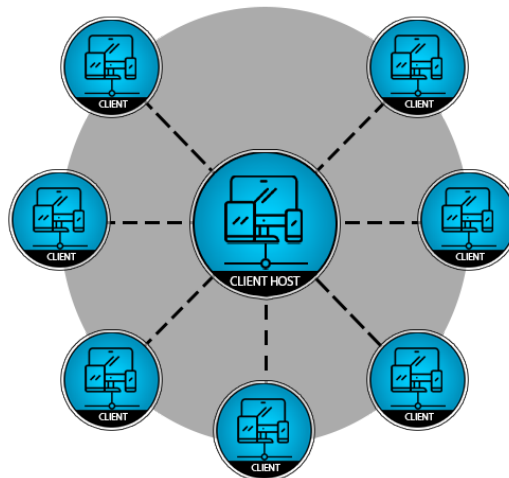

Figura 4.2: Topología cliente host

conectarse al servidor con programas cliente separados para poder ver e interactuar con el juego.

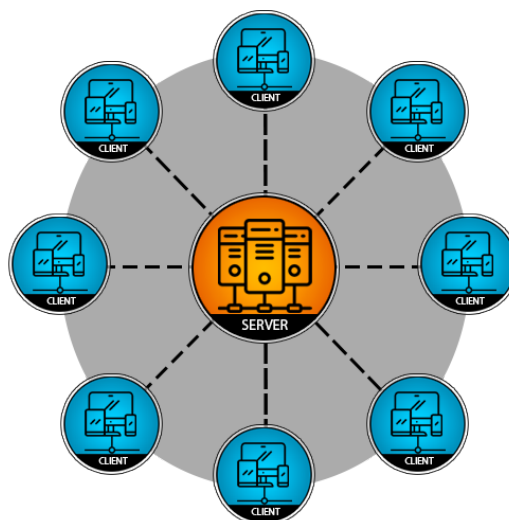

Figura 4.3: Topología de servidor dedicado

En el caso de este proyecto en concreto se ha querido adoptar la topología en la cual un cliente hace la función de host debido a que es de la que más documentación se ha encontrado y la que menos costes supone, ya que no ha sido necesario disponer de ningún servidor.

4.3 Sincronización del movimiento

Se quiere sincronizar el movimiento de ambos personajes de la forma más eficiente posible y con el menor tiempo de retraso viable para que ambos jugadores puedan ver en su pantalla como el personaje del otro jugador se mueve por el escenario y ambos puedan disfrutar de una experiencia de juego satisfactoria.

Para conseguir esto uno deberá ser el host y el otro conectarse a la partida del host como cliente, y para ello hará falta un adecuado intercambio de mensajes entre el cliente y el host.

Otro punto importante es que ambos jugadores no deben colisionar entre sí cuando estén recorriendo el nivel, deben ser capaces de pasar por la misma posición sin chocarse para no entorpecer la mecánica del juego.

4.4 Sincronización de las animaciones

Al igual que se ha mencionado en el apartado anterior, uno de los objetivos del proyecto es lograr que las animaciones de ambos personajes estén sincronizadas para que el personaje ejecute la animación adecuada, por ejemplo, al saltar, y el otro jugador pueda ver como su compañero salta. Esto se requiere hacer, al igual que con el movimiento, de forma eficiente y con el menor tiempo de retraso, con un óptimo intercambio de mensajes.

4.5 Enemigos

Se requiere que los enemigos solo aparezcan una vez para ambos usuarios y también que estén sincronizados para ambos jugadores, esto es, que cuando un jugador mata o ataca a un enemigo, el otro pueda ver como esto sucede también en su pantalla. En resumen, se tienen que comportar como *objetos de red*.

Otro punto importante a tener en cuenta es que el comportamiento del enemigo debe funcionar con ambos jugadores, es decir, debe ser capaz de detectar a ambos jugadores e ir a atacar al más cercano y que su máquina de estados funcione por igual al detectar cualquiera de los dos personajes.

4.6 Objetos

A lo largo de los niveles, los jugadores tendrán que usar una serie de elementos disponibles para progresar, elementos a los que se puede acceder a través de la interfaz del inventario, que analizaremos más adelante. Dichos elementos los obtendrán dentro de diferentes cofres repartidos por los escenarios. Tanto los cofres como los objetos deberán ser *objetos de red* y tienen que estar sincronizados para que ambos jugadores puedan abrir los cofres y obtener y usar los objetos de forma independiente cada uno.

Existen los siguientes objetos:

- **Señuelo:** Cuando un jugador lo use, tanto él como su compañero, deben ser capaces de ver el camino que seguirá la trampa cuando se vaya a lanzar. Cuando se lanza, la trampa permanecerá en el pavimento de ambos escenarios durante unos segundos, atraer a los enemigos al alcance máximo para distraerlos y poder avanzar evitando el conflicto.

- **Espada:** Necesaria para poderse enfrentar a los enemigos.
- **Bola de fuego:** Se utiliza para encender antorchas, que también deberán ser *objetos de red* y estar sincronizadas para ambos jugadores, presentes en el entorno a lo largo del juego, haciéndolo más interactivo con el mapa.
- **Bola de agua:** : Al igual que el anterior objeto, su utilidad radica en interaccionar con las antorchas, pero para apagarlas esta vez, para que, al no haber luz, los enemigos no sean capaces de localizar a los jugadores.

4.7 Cambio de escenas

Otro apartado importante donde ha habido que realizar diversos cambios ha sido a la hora de cambiar las escenas. Ya que los objetos de ambos jugadores deberían permanecer en el inventario de cada uno sin perderse al cambiar de escena, también el juego tiene que ser persistente, esto quiere decir que al cambiar de escena no hace falta volver a empezar ningún host ni el cliente volver a conectarse, simplemente debe seguir la partida, y la escena se debe cambiar para ambos.

También se ha decidido, por facilidad, que la escena se cambiara cuando uno de los 2 jugadores llegue al final del nivel, indistintamente de cuál sea.

4.8 Cámara

También es importante destacar algunos elementos que no son tan extensos como los anteriores, pero que también son claves para brindar una experiencia de juego satisfactoria, como es la cámara del juego, ya que se debe programar do forma adecuada para que, aún solo teniendo una cámara en el escenario, siga al jugador correspondiente en cada pantalla de juego.

4.9 Interfaz

Uno de los elementos básicos de un videojuego es la interfaz. En ella se presentan varias posibilidades a los jugadores, desde el menú para empezar a jugar o desde los elementos visuales necesarios para comprender correctamente el juego. Una de las intenciones más fundamental de este proyecto es la de permitir que 2 jugadores puedan conectarse a una partida online desde el Menú Principal, uno siendo el host y el otro uniéndose a la sesión del host como cliente. Para ello se debe habilitar el Menú Principal, pon los botones, campos de texto y *scripts* que sean necesarios para permitir este comportamiento.

Otra interfaz importante que debe estar presentes en el juego y cuyo comportamiento hemos explicado en el apartado 3.2 de esta memoria es la del menú de pausa. También habrá presentes más menús en el juego, como es el de victoria, que se mostrará al acabar cada nivel y que permitirá a los jugadores pasar al siguiente nivel; debe poder hacer esto sin que se cierre la conexión con el servidor, para que los usuarios puedan seguir jugando sin tener que volver a iniciar una nueva conexión y los elementos de los inventarios también deben permanecer intactos. Todo esto con el objetivo de tener una experiencia de juego más satisfactoria y fluida. Otra interfaz que tiene que estar presente es la del menú de muerte, que consiste en permitir al usuario volver a empezar el juego desde el inicio.

Otros elementos pertenecientes a la interfaz de dentro del juego que habrá que adaptar para que funcionen de forma óptima en un juego online son los inventarios para cada jugador, porque cada uno de ellos debe tener un inventario independiente al de su compañero y deben ser capaces de acceder a él de forma independiente y asimismo de usar los diferentes objetos de forma separada respecto al otro jugador.

Por último, otro componente que forma parte de la interfaz interna, es la vida de cada jugador, que está representada por 3 corazones, y se va restando medio corazón cada vez que se recibe un ataque de un enemigo. Es imperativo hacer que este elemento funcione por separado en cada jugador, cuando se resta vida, que se disminuya únicamente al jugador que corresponde y que se represente de forma correcta en la interfaz del jugador correspondiente.

CAPÍTULO 5

Desarrollo de la solución

Este capítulo está plenamente centrado en la implementación de las soluciones que se han llevado a cabo a las situaciones descritas en el capítulo anterior. Se desglosará esta sección en cada uno de los elementos que se han implementado o modificado para este proyecto. Además, se explicará el proceso implementado y el software detrás de cada uno. También han ido surgiendo a lo largo del desarrollo de este proyecto una serie de dudas que han sido resueltas con ayuda de diversos foros por Internet, documentos oficiales de Unity y documentación externa [13] [14], así como una serie de proyectos implementados por otros desarrolladores [11] [12], [15].

5.1 Topología de red

La arquitectura servidor de escucha/host tiene la característica de que un jugador puede alojar el juego en su propia máquina, lo que significa que el host también es un jugador, tiene la ventaja de que es libre y no requiere una arquitectura de red especial, ya que los jugadores suelen estar en la misma *LAN*. Sin embargo, conectarse a los PCs de otros usuarios puede ser más complejo de lo esperado, debido a que los equipos locales están detrás de *NATs* y routers. Por suerte, existen varias arquitecturas que simplifican este proceso, como *Port Forwarding*, *NAT Punchthrough* y *Relay Servers* (servidor de retransmisión).

Para implementar este tipo de arquitectura, Unity proporciona una solución confiable mediante un *Relay Server*. Esta opción nos reenvía automáticamente los puertos y permite que el servidor de escucha también se ejecute en la nube; sin embargo, puede haber costos adicionales, ya que estamos usando la infraestructura de otro.

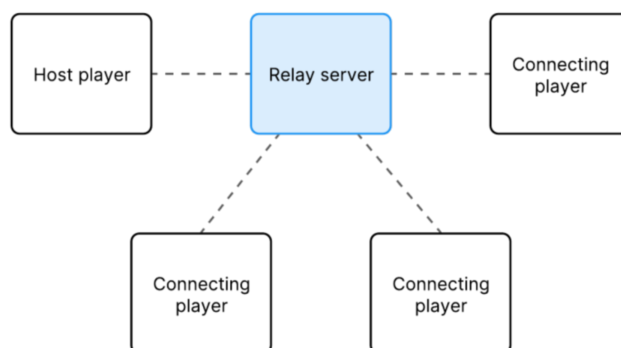

Figura 5.1: Arquitectura usando Relay Server

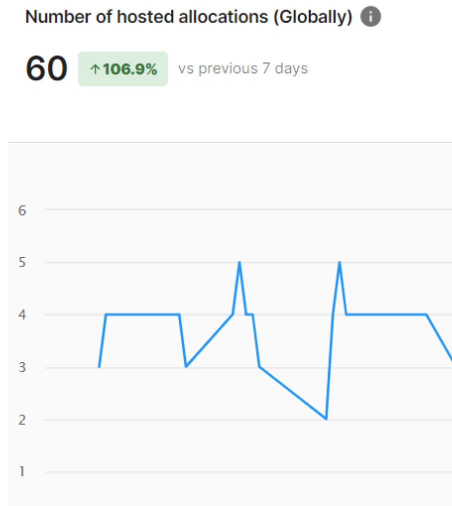

Figura 5.2: Número de asignaciones al host en nuestro Relay Server

Antes que nada, debemos instalar el paquete de Unity de *Netcode*, que como ya hemos explicado en el capítulo anterior, es una librería de redes que consigue aislar la red del proyecto, favoreciendo así que el desarrollador se centre más en el proyecto que en los marcos y protocolos de red. Para hacer que el *Relay Server* funcione, hemos tenido que habilitar los *Unity Services* para este proyecto, instalar en él el paquete de *Relay Server* y activar el servicio de Relay en la web.

Seguidamente, se ha creado un prototipo con una escena base y en ella hemos agregado un componente *NetworkManager*, que es un *script* cuya función es la de manejar todas las configuraciones relacionadas con la red. En este mismo componente debemos especificar cuál es el *prefab* del protagonista para que cada vez que un cliente se conecte al juego, el Manager se encargue de crear una nueva instancia de dicho *prefab* en la escena. En este componente también se requiere especificar que el tipo de transporte a través de la red será *Unity Transport*. En dicho componente cabe especificar que el tipo de protocolo será *Relay Unity Transport*, lo que permite hacer uso del *Relay Server*.

El siguiente paso será programar la conexión, puesto que con el *Relay Server* no se hace de forma automática. Para ello hemos creado un *script*, basándonos en la documentación oficial de Unity, que nos servirá para hacer la conexión de los clientes y para generar un código de acceso que tendrá que introducir el cliente al conectarse. Esta clase la agregaremos a un objeto vacío que debe estar presente en las escenas de los niveles.

En adición a esto debemos crear otras 2 clases, una que contendrá los datos del host/servidor (como la dirección IP, el puerto, el código de acceso...) y otra con la información del cliente.

Por otro lado, cada objeto que necesita ser replicado a través de la red debe tener un componente *NetworkObject*, así que fue añadido al *prefab* del protagonista. Con esto, al ejecutar el juego, ya se puede empezar el host desde el objeto de *NetworkManager* que se ha añadido a la escena y, por lo tanto, nuestra protagonista aparecerá y nos podremos mover con ella por la escena base.

Si se le echa un vistazo al componente *NetworkObject* de la protagonista mientras se está ejecutando el Host, se pueden observar propiedades muy interesantes como la variable *IsOwner*. Dado que dos jugadores pueden conectarse a la escena, se creará más de un *prefab* de la protagonista; sin embargo, esta propiedad solo será verdadera en aquellos objetos de juego creados por un cliente específico y falso en el otro cliente. Este hecho es

importante cuando se altera el estado de otros objetos en la red y será de gran utilidad en los futuros pasos del proyecto.

Es importante explicar también, que todas las clases necesarias para que la topología de red funcione deben ser *NetworkBehavior*, para que puedan usar *NetworkVariables* (explicadas más adelante) y RPCs (llamadas a procedimientos remotos) para sincronizar el estado y enviar mensajes a través de la red. *NetworkBehaviour* es una clase abstracta que se deriva de *MonoBehaviour* y se usa principalmente para crear una lógica de juego/código de red única. Cuando llama a una función RPC, la función no se llama localmente. En su lugar, se envía un mensaje que contiene sus parámetros: el *networkId* del *NetworkObject* asociado con el *GameObject* al que está asignado *NetworkBehaviour* y el "índice"del *NetworkObject* relativo *NetworkBehaviour* (es decir, un *NetworkObject* podría tener varios *NetworkBehaviours*).

5.2 Interfaz

Este apartado trata de cómo se ha implementado la interfaz, tanto de los menús como la propia interna del juego, para que pueda funcionar de forma correcta en nuestro juego online y actúe de forma sincronizada para ambos jugadores.

Lo primero que se ha desarrollado ha sido la interfaz del Menú principal. Como ya hemos explicado en el apartado 3.2, consta de los botones 'Crear partida online', 'Opciones' y 'Salir'. Una vez dentro del menú de 'Crear partida online' se ha creado un botón 'Crear partida' para iniciar el host, un campo de texto para que el otro jugador pueda enviar el código para unirse a la sesión, un botón para unirse una vez puesto el código y el botón de jugar, que hace que esta interfaz desaparezca y podamos ver a los personajes dentro del nivel.

También se ha añadido a la interfaz un área de *debug*, donde aparecen los jugadores que se han conectado, cuando lo han hecho y también imprime el código de conexión que debe introducir el cliente para conectarse.

Para que todo esto funcione de forma correcta se ha implementado una clase *Singleton*, que nos permite asegurarnos de que el resto de clases clase tengan una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia, la clase *Logger*, que se encarga de imprimir la información que se ha comentado en el área de *debug*. Y por último la clase *UIManager*, que es la responsable de comunicarse con el *NetworkManager* y con el *RelayManager* para que inicie el host o el cliente, dependiendo de cada botón, y para llamar a los métodos de *Logger* para imprimir la información cuando se inicia el host o se conecta un cliente, etcétera.

Figura 5.3: Interfaz Menú Principal y Crear Partida Online

En la interfaz interna del juego nos encontramos con el inventario, el cual se requirió mover dentro del *prefab* de la protagonista para que se comportara de forma independien-

te para cada jugador y que cada cliente pudiera acceder al suyo. Para que cada jugador pudiera acceder a sus respectivos objetos dentro del inventario de utilizarlos, se debe hacer la comprobación de que sea el propietario (IsOwner) y cliente. También se ha tenido que introducir un método para recoger los objetos dentro del *script* del inventario para hacer esta misma comprobación.

En cuanto a la vida de cada jugador se ha hecho algo muy parecido que con el inventario, se debe introducir dentro del *prefab* de la protagonista y se han hecho también la misma comprobación de que sea propietario y cliente para que la vida baje de forma independiente en cada personaje y se vea reflejado en la interfaz que muestra la vida de cada uno.

Figura 5.4: Los 2 jugadores conectados

5.3 Movimiento

En esta sección se explicará como se ha codificado los conceptos básicos para permitir que el jugador se mueva y sincronice sus movimientos en todos los nodos. Para ellos hemos cogido el código del movimiento de la guerrera, en el cual se encuentran todas las funciones relacionadas con la posición del jugador, tanto su propio movimiento como la capacidad de ocultarse, subir o bajar escaleras, saltar, agacharse y embestir, separadas en su propio método, y lo hemos adecuado de forma que los clientes puedan ver el movimiento de los otros jugadores en su pantalla.

Esto se conseguirá creando 2 *NetworkVariables* una de ellas controlará el movimiento en el eje X y la otra en el eje Y. Una *NetworkVariable* es una variable con su valor rastreado por el *SDK* de *Netcode*. Sus valores se replican a otros clientes conectados al servidor regularmente. Cuando un cliente se conecta inicialmente a un host, todos los últimos .estados"de los valores relevantes de las *NetworkVariables* se replicarán en ese nuevo cliente y sus estados se actualizan a intervalos regulares.

También se han creado 4 nuevos métodos para que todo funcione correctamente:

- **UpdateServer():** Es el método que ejecuta el servidor, ya que el movimiento real se produce en el servidor y los valores se sincronizan entre los nodos. Este método actualiza las posiciones del jugador.
- **UpdateClient():** Es el método que ejecuta el movimiento real que harán las protagonistas, sin embargo, los clientes no pueden modificar directamente las Network-Variables, debido a que los permisos predeterminados especifican que el servidor

es el único que puede modificarlas, por lo tanto, en este método también se le pide al servidor que las actualice. Este método se ejecuta si se cumplen las condiciones de *IsOwner* y *IsClient*.

- **UpdateClientPositionServerRpc(float xMovement,float yMovement):** Se trata de un método especial que termina con el sufijo *ServerRpc* y se anota como [ServerRpc]. Esto es necesario cuando un cliente necesita un servidor para ejecutar un código y sirve para actualizar las *NetworkVariables* de movimiento que hemos creado.

- **private void StopMovementFromClientServerRpc():** Se vuelve a tratar de un método *ServerRpc* y se ejecuta cuando se quiere dejar de mover las posiciones de los clientes cuando no se está presionando ninguna tecla, haciendo que la variable *xMovement* sea igual a 0.

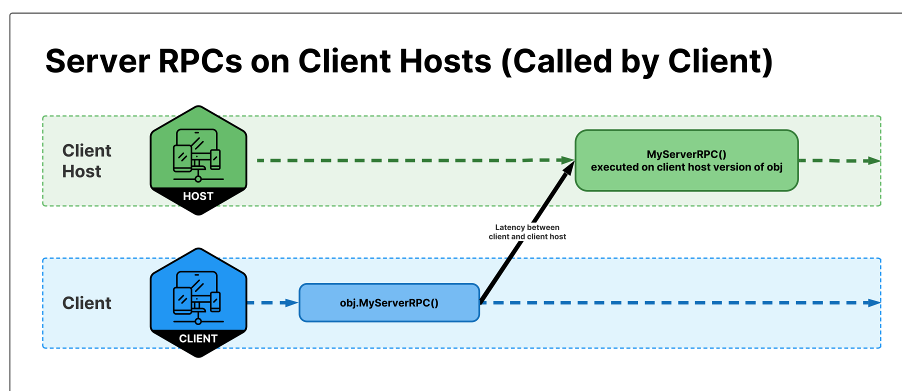

Figura 5.5: Llamada un cliente a un método ServidorRPC

5.4 Animaciones

En este apartado se va a explicar como se han sincronizado las animaciones de las protagonistas a través de la red. El paso inicial es agregar el componente especial *NetworkAnimator* al *prefab* del personaje. El componente *NetworkAnimator* sincroniza animaciones a través de la red. Al usar *NetworkAnimator*, solo el propietario debe actualizar el estado del componente *Animator* vinculado a *NetworkAnimator*. El compañero recibirá automáticamente actualizaciones del estado de la animación.

Para ello se ha creado otra *NetworkVariable* en la cual se almacena el estado de la protagonista, ya sea correr, saltar, agacharse, etcétera. Esta variable, junto con un método que hemos nombrado *UpdateAnimation()*, se encargan de definir en que estado se debe encontrar la protagonista, dependiendo de las teclas que se pulsen, y de ejecutar la animación correspondiente. También se ha tenido que crear otro método *ServerRPC* llamado *UpdatePlayerStateServerRpc(PlayerAnimation newState)* que se ejecuta cuando el cliente necesita que el servidor actualice el estado de la animación del personaje al estado que se solicita.

5.5 Enemigos

Esta sección trata de como se ha conseguido sincronizar a los personajes enemigos entre ambos jugadores, para que ambos puedan atacarlos y acabar con ellos, y cada uno pueda ver en su pantalla las acciones de los enemigos respecto a ellos mismos y al otro jugador. También ha sido importante añadir la funcionalidad de que los enemigos puedan

detectar a ambos personajes y que decidan perseguir y atacar al que encuentren más cerca, cambiando su comportamiento en caso de que el otro jugador se sitúe en una posición más cercana respecto al que ya estaban persiguiendo.

Conseguir que objetos, o en este caso los enemigos, se sincronicen a través de la red, es bastante complejo, ya que sincronizar valores reales con suficiente precisión por la red presenta una dificultad muy alta. Para lograr esto, Unity ha desarrollado algunos componentes que permiten la sincronización de *rigidbodies* con relativa facilidad.

Lo primero que debemos hacer dado que queremos mantener sincronizado todo lo que sucede con la física de los enemigos, debemos agregar un componente *NetworkRigidbody* a todos los *prefabs* de los adversarios, esto agregará automáticamente un componente *NetworkTransform*. Finalmente, añadiremos también el elemento *NetworkObject*.

Para notificar a *NetworkManager* que se pueden crear instancias de otros elementos debemos agregar los *prefabs* a la propiedad *NetworkPrefabs* del *NetworkManager*.

El siguiente paso será generar estos enemigos en la escena y para eso se ha creado un *script* llamado *SpawnManager* que añadiremos a un objeto vacío en las escenas. En este *script* el servidor se encarga de crear los enemigos recorriendo un bucle, y se llama al método *Spawn()* de *NetworkObject*. A continuación indicamos los *prefabs* de los enemigos que se tienen que generar y añadimos a este mismo elemento vacío el componente *NetworkObject*, ya que es de tipo *NetworkBehaviour*. Finalmente, modificamos el código de *UIManager* para que al iniciar el host se generen a su vez los enemigos. De esta forma ya estarán sincronizados en ambos jugadores.

Figura 5.6: Sprites de los enemigos

5.6 Cambio de escenas

Este apartado trata de como se ha conseguido que el servidor siga activo al cambiar de nivel y, por tanto, los jugadores puedan seguir conectados y jugando al juego sin tener que iniciar otra sesión. A la vez, también se busca que los inventarios de cada personaje permanezcan intactos al cambiar la escena.

Para poder cambiar de escena, el *Scene Manager* normal que viene con Unity no nos sirve, puesto que al cambiar de escena se cierra el servidor. Por tanto, se ha tenido que usar el *NetworkSceneManager* que viene con *Netcode* y nos permite efectuar este cambio sin cerrar la conexión.

NetworkSceneManager proporciona un completo sistema de notificación de eventos de escena. Proporciona eventos de carga y descarga de escenas y sincronización del cliente mediante *Scene Events* que se pueden rastrear tanto en el cliente como en el servidor. El servidor es capaz de rastrear el progreso del evento de escena de cada cliente y los clientes reciben mensajes de eventos de escena del servidor, activan notificaciones locales

a medida que avanza a través de un evento de escena y envían notificaciones locales al servidor. Todas las escenas cargadas a través de *NetworkSceneManager* y todos los objetos *Netcode* generados se sincronizarán con los clientes durante la sincronización del cliente al cambiar de escena.

Para esto en el *script* del Menú de victoria hemos añadido que se cargue las escenas que correspondan con el método NetworkManager.Singleton.SceneManager.LoadScene(...).

Para que los objetos se mantengan en el inventario de ambos jugadores al cambiar de nivel, simplemente se ha creado una clase en el *prefab* del protagonista con la siguiente línea de código: dontDestroyOnLoad(this.gameObject), siendo el *GameObject* la protagonista. De esta forma, al cargar una escena nueva no se destruirá el personaje ni sus componentes, como el inventario. Esto permite también conservar la cantidad de vida que les queda a los jugadores entre los niveles.

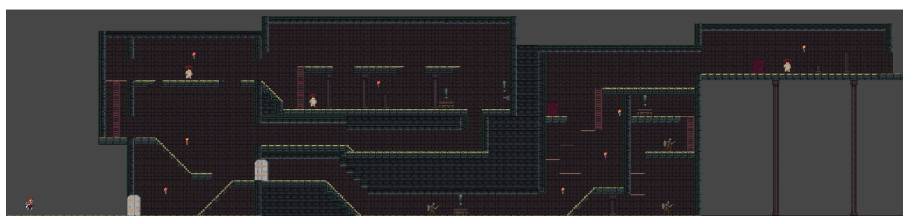

Figura 5.7: Escena del primer nivel

5.7 Cámara

En esta sección se expondrá como se ha conseguido que habiendo solo una cámara por escena siga al personaje que corresponda según la instancia del usuario que sea.

Para ello se ha hecho uso de un nuevo paquete llamado *Cinemachine*, que ayuda a crear una experiencia más inmersiva para los jugadores.

Lo primero que debemos modificar es la cámara principal, agregando los componentes *CinemachineBrain* y *CinemachineVirtualCamera*. En la propiedad Body de *CinemachineVirtualCamera*, debemos seleccionar *FramingTransposer*, que está indicada para cámaras 2D. Esto provocará que aparezca una advertencia, puesto que es necesitamos especificar que objeto del juego debe seguirse, pero no tenemos a nuestra protagonista en escena, ya que se crea desde el *prefab* en el momento de la ejecución. Es importante asegurarse también de que las propiedades *AmplitudeGain* y *FrequencyGain* estén configurados en 0, de lo contrario, la cámara se comportará de manera extraña cuando la protagonista aún no haya sido generada.

Para que la cámara pueda seguir a nuestra protagonista se ha creado un nuevo *script* y lo hemos adjuntado a la cámara principal. Este script contiene un método que se encarga de que *CinemachineVirtualCamera* detecte y siga al jugador que corresponda. Debemos modificar también el *script* del movimiento para que en el método *Start()* se llame al método de la otra clase que se encarga de seguir al jugador pasándole como parámetro un objeto vacío que hemos añadido al *prefab* de la protagonista, para que siga ese punto en concreto. De esta forma, la cámara ya detecta al personaje correspondiente de cada jugador y los sigue con la cámara en la pantalla de cada uno.

5.8 Interacción con el escenario

A lo largo de los niveles se han implementado unos componentes en forma de cortinas integradas en el mapa del nivel y se detecta donde está a través de un *trigger*. Estas cortinas sirven para que los jugadores puedan esconderse detrás de ellas y no sean vistos por los enemigos. Por tanto, este comportamiento también debe estar sincronizado entre los 2 jugadores.

Se ha conseguido sincronizar de una forma similar al movimiento, hemos creado una *NetworkVariable* que nos indica si el jugador está escondido o no, y si es propietario y cliente se ejecuta el método que se encarga de hacer que el *Sprite Renderer* se desactive para que no se vea al personaje y se desactive también el *collider* para que los enemigos no se choquen con él ni lo detecten, todo esto si se encuentra en una ubicación donde hay una cortina. Al final de este método se llama a otro método *ServerRPC* que se ha creado para que el servidor actualice la *NetworkVariable* y el otro jugador pueda percibir que su compañero se ha escondido.

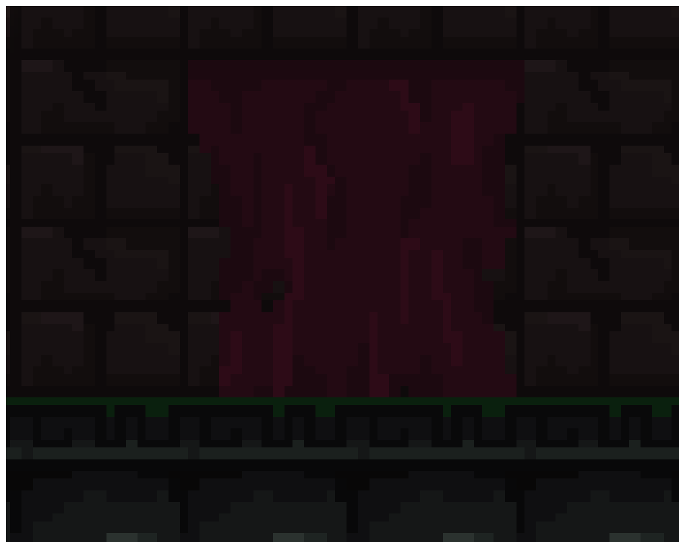

Figura 5.8: Cortina en el escenario

CAPÍTULO 6
Pruebas

Tras la finalización del desarrollo del proyecto del juego y sus pruebas concurrentes y con una base funcional para un solo jugador, se pasó a las pruebas Alpha y Beta del sistema online para comprobar su funcionamiento y cómo los distintos sujetos que participaron en estas pruebas interactúan con los sistemas implementados.

Además, estas pruebas se ven separadas en 2 pruebas distintos agrupados bajo el nombre de Alpha y Beta y que implican un cambio en las versiones y un salto entre las pruebas internas entre los miembros integrantes del proyecto y unas pruebas externas sujetas a las características descritas a continuación.

Para ello se eligió un abanico de personas de distintos conocimientos en torno a videojuegos para ver si el sistema era funcional sin importar la experiencia previa en videojuegos del mismo estilo o con sistemas parecidos.

6.1 Pruebas Alpha

Durante la primera serie de pruebas internas en el equipo se realizaron un conjunto de mejoras relacionadas con los sistemas raíz:

La sincronización presentaba un error que durante una larga sesión de juego se empezaba a acumular un retraso en dicha sincronización de los personajes que hacía que los movimientos de un jugador y otro no ocurriesen a la vez en ambos jugadores, tras unas cuantas pruebas se encontró el error y se soluciona rápidamente.

Otro problema encontrado a lo largo de estas pruebas fueron que las animaciones de los personajes no siempre se mostraban o sincronizaban con los movimientos realizados por cada jugador, este error también se pudo solucionar en una fase temprana gracias a estas pruebas.

6.2 Pruebas Beta

Durante estas pruebas y con los errores y el sistema más pulidos, las mejoras realizadas fueron menores y más sujetas a preferencias de los usuarios que las realizaron e indicaron algunas mejoras poco relevantes como algunos cambios poco significativos en las interfaces y algunos problemas como que la vida de los personajes en ocasiones no se mostraba en la interfaz de la forma requerida incluso aunque el jugador sí la perdiese.

CAPÍTULO 7
Conclusiones

A lo largo de la elaboración de este proyecto se reforzaron las nociones que ya se habían adquirido previamente respecto al desarrollo de videojuegos. En el pasado, ya se habían llevado a cabo diferentes modelos y proyectos haciendo uso de Unity, así como lo que se podría calificar como un nivel de un juego de realidad virtual y otro pequeño juego con un par de niveles en dos de las asignaturas de este curso. Sin embargo, a pesar de que ya se tenía cierta base, se ha requerido estudiar y profundizar sobre las amplias y diversas funcionalidades que ofrece Unity a la hora de desarrollar videojuegos multijugador, ya que no se conocían hasta el momento.

Por lo que respecta a los objetivos que se han propuesto para este proyecto y que están expuestos al inicio de esta memoria, se puede afirmar que todos ellos se han cumplido y están documentados, contando con los otros dos trabajos de final de grado de los miembros que componen el grupo del proyecto. El producto final de este proyecto ha sido un videojuego de sigilo y plataformas cooperativo, que ha sido el más importante de los objetivos de cara a este TFG, permitiendo a los jugadores conectarse a una sesión conjunta, saltar entre las diferentes plataformas y ocultarse de la presencia de los adversarios. En cuanto al trabajo que se ha hecho en este TFG en concreto:

Se ha logrado la posibilidad de que el juego pueda disfrutarse en modo cooperativo con un amigo, usando una topología de red en la cual uno de estos jugadores, aparte de ser cliente, también es el host del juego y haciendo uso de un servidor de escucha en la nube que permite que estas conexiones entre clientes sean posibles. También se ha logrado gracias al uso de los diferentes paquetes mencionados en esta memoria.

Cabe afirmar que, a lo largo del desarrollo, han surgido ciertos conflictos con los paquetes usados, ya que las redes multijugador en Unity es una materia que va avanzando de forma muy rápida debido a aparición de mejoras y, por tanto, había mucha información y versiones de los paquetes que estaban desfasados, lo cual ha causado que ciertos elementos en los que se han trabajado hayan tenido que volver a plantearse a lo largo del desarrollo.

CAPÍTULO 8
Relación con los estudios

Es importante señalar que el desarrollo de este videojuego está muy relacionado con dos asignaturas que se estudiaron este curso. Una de ellas es **Introducción a la Programación de Videojuegos**, en la cual se realizaron numerosos proyectos utilizando Unity y donde se aprendieron conceptos básicos sobre como programar el movimiento, como crear escenas, como gestionar la iluminación, los sonidos, las interfaces, contadores, entre otros. Además, en esta asignatura se llevó a cabo como proyecto final un pequeño juego básico con un par de niveles. La otra asignatura fue **Realidad Virtual y Aumentada**, en la cual, aparte de aprender gran cantidad de información sobre como funcionan y han ido evolucionando estas tecnologías, se creó también un proyecto en Unity de realidad virtual en el cual se trabajaron bastante las físicas de Unity, los escenarios y los objetos.

Por otro lado, a lo largo de la carrera se han cursado asignaturas que han ayudado significativamente a obtener las competencias necesarias que han permitido la realización con éxito de este proyecto, como han sido **Programación** e **Introducción a la Programación** ya que establecieron las bases de los lenguajes de programación, aunque no se tratara C# específicamente (el lenguaje que utiliza Unity). También se han obtenido muchos conocimientos de redes gracias a haber cursado la mención Tecnologías de la Información con asignaturas como **Diseño y Configuración de Redes de Área Local**, **Redes Corporativas**, **Sistemas y Servicios en Red** y **Administración de Sistemas** ente otras, que han permitido que se pueda comprender con más facilidad como gestionar la topología de red escogida para el proyecto y como funcionan las redes dentro de Unity.

CAPÍTULO 9

Trabajos futuros

A continuación, a pesar de la implementación satisfactoria del online al proyecto, se exponen algunas áreas en las que en caso de haber dispuesto de una mayor cantidad de tiempo se podrían haber hecho mejoras. Entre ellas, algunas harán referencia a cosas que se podrían añadir para ampliar la funcionalidad e incluso el juego.

- Se podría implementar la posibilidad de separar a los jugadores en distintos niveles y que necesiten colaborar mediante la creación de nuevas mecánicas, era una posibilidad que se quería explorar con nuevas mecánicas que por falta de tiempo no se llegaron a implementar.
- Un sistema de tiempos en el que se pudiese llevar un registro de cuán rápido se ha pasado un jugador los niveles también es un objetivo a futuro.
- Cambiar la apariencia del personaje del segundo jugador, para que no sean los dos idénticos.
- Permitir que más jugadores se puedan conectar al juego y tengan diferentes personajes.

CAPÍTULO 10
Glosario

- **Animator:** El componente Animator es utilizado para asignar una animación a un *GameObject* en su escena.
- **Arcade:** Tipo de videojuego creado para ser jugado en salones recreativos.
- **Asset:** Cualquier archivo que se encuentra guardado dentro del proyecto y puede ser usado en el mismo.
- **Backlog:** Lista que contiene el conjunto de actividades a desarrollar para un proyecto.
- **Cartuchos ROM:** Los cartuchos *Read Only Memory* son un método de almacenamiento utilizado para cargar software como videojuegos u otros programas de aplicación.
- **Collider:** Componente que se incorpora a los objetos en los que se desea detectar colisiones con otros objetos.
- **Componente:** Elemento que se añade a un *GameObject* para añadir funcionalidad.
- **Comunicaciones UDP:** Es un protocolo que permite la transmisión sin conexión de datagramas en redes basadas en IP.
- **Diagrama de Gantt:** Gráfica en la que se presenta la organización y tiempos estimados en el desarrollo de un proyecto.
- **Escena:** Contiene los entornos del juego y los menús, se debe pensar en una escena como si fuera un nivel.
- **Feedback:** Término inglés que puede traducirse como realimentación o retroalimentación. En esta memoria se refiere a la opinión que ha sido aportada por las personas que han probado el videojuego.
- **Game Object:** Objetos fundamentales en Unity que representan personajes, objetos, y el escenario. Estos no logran nada por sí mismos, pero funcionan como contenedores para *Componentes*, que implementan la verdadera funcionalidad.
- **Host:** Las computadoras u otros dispositivos (tabletas, móviles, portátiles) conectados a una red que proveen y utilizan servicios de ella.
- **Indie:** Aquellos juegos creados por empresas independientes, normalmente mediante un presupuesto y equipos de desarrollo reducidos.

- **Monobehaviour:** Clase base de la cual heredan los *scripts* de Unity. Esta clase solo podrá ser heredada en *scripts* situados en objetos presentes en la escena y permitirá añadir varias funcionalidades como los métodos *Start()* o *Update()* entre otros.
- **NAT:** Permite que redes de ordenadores utilicen un rango de direcciones especiales (IPs privadas) y se conecten a Internet usando una única dirección IP (IP pública).
- **NetworkObject:** Componente que se agrega a los objetos que queremos que sean sincronizados o replicados por la red entre los dos jugadores.
- **NetworkTransform:** El componente *NetworkTransform* sincroniza el movimiento y la rotación de *GameObjects* en la red.
- **Online:** Utilizado para referirse a un juego multijugador a través de Internet.
- **P2P:** Es un protocolo de red de ordenadores en la que todos o algunos aspectos funcionan sin clientes ni servidores fijos, sino una serie de nodos que se comportan como iguales entre sí. Es más, actúan simultáneamente como clientes y servidores respecto a los demás nodos de la red. Las redes P2P permiten el intercambio directo de información, en cualquier formato, entre los ordenadores interconectados.
- **Port Forwarding:** El port forwarding es una forma de redirigir conexiones a través del router. Al habilitar el port forwarding, se permitir que un dispositivo remoto conecte de forma directa con un equipo local a través de un redireccionamiento que se realiza designando puertos específicos del router para establecer esta conexión.
- **Prefab:** *Asset* que permite almacenar un *GameObject* completamente, con sus componentes y propiedades. El *prefab* actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un *prefab* será inmediatamente reflejado en todas las instancias producidas de él.
- **Rigidbody:** Componente que permite a los *GameObject* actuar bajo el control de las físicas. Puede recibir fuerzas para actuar de una forma más realista.
- **Script:** Fragmento de código encargado de añadir o realizar funciones.
- **SDK:** Kit de Desarrollo Software, es un conjunto de herramientas proporcionado usualmente por el fabricante de una plataforma de hardware, un sistema operativo (SO) o un lenguaje de programación (en este caso Unity).
- **Shooter:** Género de videojuegos de acción donde el principal objetivo es disparar y matar enemigos, generalmente con armas de fuego.
- **Singleton:** Patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.
- **Sprite:** Objetos gráficos en 2D que se utilizan para personajes, accesorios, proyectiles y otros elementos de los videojuegos 2D.
- **Sprite Renderer:** El componente que permite a mostrar imágenes como *Sprites* para su uso en escenas.
- **Trigger:** Propiedad activable en un *Collider* que permite darle un nuevo uso. En caso de estar activado, el *Collider* no reaccionará a las colisiones, pero si las detectará, siendo esto utilizable mediante código.
- **Unity Asset Store:** : Portal de Unity en el que se pueden descargar o comprar diferentes paquetes de Assets.
- **UT:** Cada una de las tareas en las que se divide el proyecto en la metodología ágil.

Bibliografía

[1] **La industria del videojuego facturó en 2020, en todo el mundo, más que el cine y los deportes juntos en EEUU** Pedro Herrero [MeriStation] https://as.com/meristation/2020/12/26/noticias/1608992024_963325.html

[2] **The ultimate history of videogames**. Steven L. Kent [Crown; Illustrated edición].

[3] **Video Games Help People to Connect and Engage During COVID-19**. Raisa Santos [Health Policy Watch] https://healthpolicy-watch.news/video-games-helps-people-to-connect/.

[4] **The Psychology of Video Games (The Psychology of Everything)** Celia Hodent [Routledge, 1ª edición, 2020]

[5] **Before the Crash: Early Video Game History**. Mark J. P. Wolf [Wayne State University Press, 2012].

[6] **Unity. Aprende a desarrollar videojuegos**. Carlo I. López Sandoval [RC Libros, 1ª Edición, 2019]

[7] **Unity Multiplayer Resources**. Unity. [En línea] https://docs-multiplayer.unity3d.com/netcode/current/learn/introduction/index.html.

[8] **Netcode for GameObjects**. Unity. [En línea] https://docs-multiplayer.unity3d.com/netcode/current/about/index.html.

[9] **Unity Relay**. [Unity] https://docs.unity.com/relay/introduction.html.

[10] **Unity Asset Store**. Unity. [En línea] https://assetstore.unity.com/?gclid=CjwKCAjwquWVBhBrEiwAt1KmwuublLSALWo-KSP88ml3h_BhiB3XjW0Vo7ZI26n-3qZ4MtjBspqRwxoCh40QAvD_BwE&gclsrc=aw.ds.

[11] **Brackeys**. Github. [En línea] https://github.com/Brackeys.

[12] **CEFIRE 2022 - Multiplayer**. Juan Jesús Izquierdo Domenech [En línea] https://juan-jesus-izquierdo-domenech.gitbook.io/cefire-2022-multiplayer/.

[13] **Gamer to Gamer Developer**. [En línea] https://www.gamertogamedeveloper.com/.

[14] **SocketWeaver**. [En línea] https://docs.socketweaver.com/.

[15] **Dilmer Valecillos**. Github. [En línea] https://github.com/dilmerv/UnityMultiplayerPlayground.

APÉNDICE A
Controles

Aquí se exponen los controles principales a la hora de jugar al videojuego.

	Teclas
Movimiento	A / D
Salto	Espacio
Agacharse	C
Embestida	Mayús
Escaleras de mano	W / S
Bajar de escaleras o plataformas de madera	Mantener S
Interactuar // Esconderse	E
Usar objeto	Click izquierdo
Inventario	1-6
Menú de pausa	ESC

Tabla A.1: Controles

APÉNDICE B
Código relevante

En este apéndice se comenta como se ha gestionado parte del movimiento, exponemos aquí dos de los métodos ultilizados. Cabe destacar que _xMovement e _yMovement son NetworkVariables.

MovementHandler():

```
private void MovementHandler(){
Debug.Log(Input.GetKey(KeyCode.Space));
Boolean d = Input.GetKey(KeyCode.S);
if (IsGrounded() && !IsDucked() && !isHidded && Input.GetKey(KeyCode.Space) &&
!Input.GetKey(KeyCode.S))
{
    RaycastHit2D rc = Physics2D.BoxCast(colliderCabeza.bounds.center,
    colliderCabeza.bounds.size * 0.5f, 0f, Vector2.up, .15f, terrainLayerMask);
    if (rc.collider == null)
    {
        yMovement = jumpSpeed * Time.deltaTime;
        rb.velocity = new Vector2(rb.velocity.x, jumpSpeed * Time.deltaTime);
        animator.SetTrigger("Jump");
        string audioFile = jumpAudio[Random.Range(0, jumpAudio.Length)];
        AudioManager.instance.PlaySound(audioFile);
    }
}

if (Input.GetKey(KeyCode.A) && !isHidded && !dashing){
    if (GameObject.FindWithTag("Stairs") != null) {
        GameObject.FindWithTag("Stairs").GetComponent<PolygonCollider2D>().enabled
        = true;
        GameObject.FindWithTag("SCol").GetComponent<PolygonCollider2D>().enabled
        = false;
    }
    capsCol.enabled = true;
    boxCol.enabled = false;
    if (lookingRight) {
        this.transform.localScale = new Vector3(transform.localScale.x *
        -1, transform.localScale.y, transform.localScale.z);
        shooter.transform.localScale = new Vector3(transform.localScale.x *
        -1, transform.localScale.y, transform.localScale.z);
    }
```

```
            lookingRight = false;
            if(IsGrounded() && !IsDucked()){
                xMovement = -movementSpeed * Time.deltaTime;
                isMoving = true;
            } else if(IsDucked()){
                isMoving = true;
                xMovement = Mathf.Clamp(-movementSpeed * duckedSpeed * Time.deltaTime,
                -movementSpeed, +movementSpeed) * Time.deltaTime;
            } else{
                isMoving = false;
                xMovement += -movementSpeed * midAirControl * Time.deltaTime;
                xMovement = Mathf.Clamp(xMovement, -movementSpeed, + movementSpeed) *
                Time.deltaTime;
            }
        }

        if(Input.GetKey(KeyCode.D) && !isHidded && !dashing){
            if(GameObject.FindWithTag("Stairs") != null) {
                GameObject.FindWithTag("Stairs").GetComponent<PolygonCollider2D>().enabled
                = true;
                GameObject.FindWithTag("SCol").GetComponent<PolygonCollider2D>().enabled
                = false;
            }
            capsCol.enabled = true;
            boxCol.enabled = false;
            if (!lookingRight) {
                this.transform.localScale = new Vector3(transform.localScale.x *
                -1, transform.localScale.y, transform.localScale.z);
                shooter.transform.localScale = new Vector3(transform.localScale.x *
                -1, transform.localScale.y, transform.localScale.z);
            }
            lookingRight = true;
            if(IsGrounded() && !IsDucked()){
                isMoving = true;
                xMovement = movementSpeed * Time.deltaTime;
            } else if(IsDucked()){
                isMoving = true;
                xMovement = Mathf.Clamp(movementSpeed * duckedSpeed * Time.deltaTime,
                -movementSpeed, +movementSpeed) * Time.deltaTime;
            } else {
                isMoving = false;
                xMovement += movementSpeed * midAirControl * Time.deltaTime;
                xMovement = Mathf.Clamp(xMovement, -movementSpeed, +_xMovement) *
                Time.deltaTime;
            }
        }
        if( !Input.GetKey(KeyCode.A) && !Input.GetKey(KeyCode.D)) {
            isMoving = false;
            rb.velocity = new Vector2(0, rb.velocity.y);
            xMovement = 0;
            StopMovementFromClientServerRpc();
                if (OnStairs()){
```

```
            GameObject.FindWithTag("SCol").GetComponent<PolygonCollider2D>().enabled
            = true;
            GameObject.FindWithTag("Stairs").GetComponent<PolygonCollider2D>().enabled
            = false;
            boxCol.enabled = true;
            capsCol.enabled = false;

        }
    }
    if(climb){
        rb.gravityScale = -1;
        rb.velocity = new Vector2(rb.velocity.x,
        Input.GetAxisRaw("Vertical") * ladderSpeed * Time.deltaTime);
    }else rb.gravityScale = 1;

    UpdateClientPositionServerRpc(xMovement,yMovement);
}
```

UpdateClientPositionServerRpc(xMovement,yMovement):

```
[ServerRpc]
private void UpdateClientPositionServerRpc(float xMovement,float yMovement)
{
    _xMovement.Value = xMovement;
    _yMovement.Value = yMovement;

}
```