UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de un videojuego multijugador de estrategia en tiempo real con Unity y PUN

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Mirete Blanco, Alejandro

Tutor/a: Abad Cerdá, Francisco José

CURSO ACADÉMICO: 2021/2022

Resumen

El propósito de este TFG es desarrollar un videojuego de estrategia multijugador del género RTS (Estrategia en tiempo real). Un videojuego es un tipo de software complejo que requiere de la correcta aplicación de metodologías y patrones de desarrollo para poder llevarse a cabo con éxito, de esta forma la aplicación de una metodología ágil con un workflow bien definido es un requisito indispensable. Por otra parte, el género de estrategia requiere de un buen diseño que permita a los usuarios percibir la gran cantidad de información necesaria para poder utilizar la aplicación de forma satisfactoria y de un diseño capaz de responder adecuadamente a los inputs de los usuarios. El aspecto multijugador online de la aplicación posibilitará hacer mejores y más pruebas del software ya que permitirá probarlo con usuarios y recabar datos de uso.

Palabras clave: Metodología ágil, RTS, usuario, pruebas de software, multijugador

Abstract

This work aims to develop a multiplayer strategy videogame of the RTS genre (real-time strategy). A videogame is a complex piece of software that requires the proper application of software development methodologies and patterns which means that the application of an agile methodology with a well-defined workflow is required so that the project may be completed successfully. Furthermore, the strategy genre requires a good design that allows the users to easily perceive all the information that the application displays as well as to appropriately react to the users' inputs. Finally, the multiplayer aspect of the videogame will allow for better and more in-depth testing as it will allow to test the application with users and collect usage data.

Keywords : Agile methodology, RTS, user, software testing, multiplayer

Tabla de contenidos

1. Introducción

1.1 Motivación

El proceso de desarrollo de software está compuesto por una serie de prácticas consideradas estándares en la industria y que son valoradas como esenciales para realizar con éxito un proyecto de software. Esto no es diferente en el desarrollo de un videojuego ya que, aunque el objetivo de este tipo de software no sea resolver un problema o facilitar la resolución de otros procesos, como un sistema de compra online, sigue siendo un software complejo que requiere de una correcta metodología de desarrollo y de buenas prácticas para llevarlo a buen puerto.

1.1.1 Motivación personal

Desde pequeño me han apasionado los videojuegos y eso es algo que no ha cambiado con la edad, especialmente juegos de estrategia y con temática histórica o de fantasía, y siempre he pensado que me gustaría ser desarrollador de videojuegos tras terminar mis estudios. También me gusta aprender cosas nuevas de temas muy diversos y la informática, pudiendo aplicarse en prácticamente cualquier ámbito, te abre las puertas a poder estudiar y adquirir conocimientos de muchas áreas distintas. En mi opinión esto es incluso más destacable en el ámbito de los videojuegos, puesto que se pueden hacer videojuegos de absolutamente cualquier cosa, te permiten contar historias, te permiten pilotar aviones o conducir coches de carreras, te permiten vislumbrar acontecimientos históricos de formas nunca vistas y también permiten conectar a personas de todo el mundo, de distintos países y con distintas culturas.

1.1.2 Motivación profesional

No es distinto a otros sectores profesionales en el ámbito de la informática, pero el sector de los videojuegos en España crece año tras y año, y cada vez proporciona empleo a más personas. En el año 2013 había 2360 personas empleadas en el sector de los videojuegos en España, cifra que para 2019 aumentó a 7320 empleos y que se prevé que supere los 8500 empleos para el año 2023. Además, durante la crisis del COVID, el 57% de los estudios españoles de videojuegos han mantenido su plantilla y el 36% la han ampliado frente al 7% de estudios que han sufrido reducciones de plantilla[1]. Esto demuestra la robustez de este sector frente a las duras circunstancias económicas actuales, mostrándose como una buena elección de futuro profesional.

1.2 Objetivos

El propósito de este trabajo es el desarrollo de un videojuego RTS acorde a estándares modernos de desarrollo de videojuegos y cumpliendo con las metodologías y procesos utilizados de forma habitual en el desarrollo de software. Para resolver el problema en cuestión se plantean los siguientes objetivos:

- Adoptar una metodología de trabajo ágil con un backlog bien definido y acotado, así como utilizar una herramienta Kanban que permita dividir el progreso de las tareas en las distintas fases del proceso de desarrollo: diseño, desarrollo y validación. Para poder aplicar la metodología ágil de forma eficaz se debe:
    - Analizar los requerimientos de la aplicación e identificar las distintas tareas que se deben llevar a cabo durante el desarrollo.
    - Formar un backlog que contenga estas tareas, estimarlas y realizar mock-ups en caso de ser necesarios.

    - Identificar la capacidad de trabajo disponible y crear sprints de trabajo acordes con las estimaciones realizadas.
- Diseñar y desarrollar un videojuego acorde a los requerimientos de proyecto que cumpla con los estándares de desarrollo de videojuegos. Con este objetivo en mente se han definido las siguientes tareas:
    - Realizar un *game design document* que recoja adecuadamente el diseño del videojuego y sus características.
    - Identificar y aplicar las técnicas de desarrollo de software adecuadas para completar el proyecto con éxito.

1.3 Metodología

Como se ha mencionado en la sección anterior, para el desarrollo del proyecto se empleará una metodología ágil similar a Scrum, ya que no hay *product owner* ni *scrum master*, debido a la naturaleza del proyecto. Sin embargo, el desarrollo se subdividirá en sprints con una duración de dos semanas cada uno y una estimación de capacidad de trabajo de dos horas diarias. La herramienta utilizada para llevar a cabo las tareas de preparación del backlog y planificación de los sprints es hacknplan, ya que es una herramienta de gestión de proyectos diseñada específicamente para el desarrollo de videojuegos.

1.4 Estructura del documento

La estructura del trabajo es la siguiente. En primer lugar, en el capítulo 2, se discutirá el estado del arte del desarrollo de videojuegos. Se analizará en profundidad la historia de los videojuegos RTS y para finalizar el capítulo se analizará el estado del arte de los videojuegos multijugador examinando distintas arquitecturas multijugador.

En el capítulo 3 se analizará y discutirá el problema que el proyecto pretende resolver. Se mostrarán las especificaciones de requisitos recogidos en el game design document, a partir de ahora GDD, así como el modelado conceptual del proyecto y los mock-ups realizados. Luego se estudiará la seguridad de la aplicación puesto que al funcionar en línea es un requisito indispensable seguido por un análisis de propiedad intelectual debido a que en el proyecto se utilizan assets pertenecientes a terceros. Finalmente se discutirá brevemente el presupuesto del proyecto.

El capítulo 4 analizará en detalle el diseño del proyecto. Se explicarán las tecnologías utilizadas y por qué han sido escogidas frente a las alternativas disponibles. A continuación, se analizarán la arquitectura y diseño de la solución.

El capítulo 5 describirá el proceso de desarrollo del proyecto. Se comentará cómo se ha llegado a la solución final y los problemas y dificultades encontrados durante el desarrollo y cómo se han solventado. También se analizará la implementación de algunas de las partes más relevantes del sistema.

El capítulo 6 presentará las pruebas realizadas, destacando pruebas con usuarios que comprueban si la aplicación cumple con lo que esperarían potenciales usuarios.

El capítulo 7 resumirá las conclusiones del proyecto y lecciones aprendidas durante su realización, así como directrices recomendadas para futuros trabajos.

1.5 Convenciones

- Los términos del glosario aparecerán como nota a pie de página la primera vez que aparezcan en caso de no estar explicados en el texto.
- Las referencias a las figuras que formen parte del texto estarán marcadas con letra cursiva y la primera letra mayúscula (*Figura 1*).
- Los términos relativos a Unity y a PUN se escribirán con letra normal.
- Cada vez que termine un capítulo se realizará un salto de página.
- Los nombres de los videojuegos que se mencionen en la memoria se escribirán con letra cursiva, así como términos de los propios videojuegos.

2. Estado del Arte

En este capítulo en primer lugar se expondrá la historia y características principales de los videojuegos RTS y a continuación las características de los videojuegos multijugador y el cómo se adapta un videojuego RTS para multijugador.

2.1 Historia de los videojuegos RTS

2.1.1 Predecesores

Antes de la aparición de los videojuegos e incluso antes de la aparición de los primeros ordenadores ya existían numerosos juegos de estrategia. Sus orígenes se remontan al siglo XIX como juegos desarrollados por los ejércitos de distintas naciones para entrenar a sus oficiales. A lo largo del siglo XX, grupos de entusiastas crearon el género de juegos de mesa que hoy en día conocemos como "war-gaming"[2]. Estos juegos pretenden ser simulaciones de guerras y batallas con el objetivo de entretener a los jugadores, como por ejemplo el *Risk*[6], creado en el año 1950 y comercializado en 1958 y que hoy en día tiene innumerables ediciones, como *Risk Juego de Tronos* o *Risk Vikings* basados en las populares series de televisión del mismo nombre, y se sigue jugando en todo el mundo. Mientras que el *Risk* es una simulación simple, también se crearon otros juegos más complejos como *Blitzkrieg* (1965)[2] en el que el jugador tiene que manejar aspectos como la industria, armada o fuerzas aéreas. El género del war-gaming ha seguido evolucionando hasta nuestros días, con juegos nuevos cada vez más complejos dando lugar al nacimiento de franquicias como *Warhammer* con sus variantes de *Fantasy* y *40.000* y muchas más.

2.1.2 Los primeros juegos de estrategia

Con el nacimiento de los ordenadores empezaron a surgir los primeros videojuegos como el *Pong* en 1972, considerado el primer videojuego comercial de la historia. Es un juego simple que pretende imitar, al tenis en el que gana el primer jugador en alcanzar diez puntos[7]. Paralelamente también comenzaron a aparecer los juegos que se convertirían en los predecesores de lo que hoy en día conocemos como RTS.

Entre los años 1960 y 1970 comienzan a surgir los primeros juegos de estrategia. Estos eran juegos de texto sin ningún tipo de gráficos como *Civil War* de 1968[2]. El juego simula batallas de la guerra civil americana en las que el jugador debe tomar decisiones como la distribución de su presupuesto antes del combate y las tácticas a emplear durante la batalla[8]. Otro de los primeros juegos de estrategia es *Hamurabi* de 1969[2], también un juego de texto de gestión económica basado en la antigua Sumeria en el que el jugador debe repartir cultivos para alimentar a su población y adquirir más tierras mientras lidia con eventos como plagas[9].

Con la mejora en los gráficos de ordenador en los años 1980 surgieron los primeros juegos de estrategia con gráficos 2D como el *M.U.L.E.* de 1983[2], juego de gestión y colonización espacial. En el juego, entre 1 y 4 jugadores llegan al planeta de Irata con el objetivo de explorarlo y de conseguir amasar una fortuna en este nuevo mundo. Los jugadores disponen de cuatro recursos distintos para este fin, y deben conseguir parcelas de terreno en las que producir estos recursos[10]. A lo largo de la década aparecieron nuevos juegos de estrategia que se

convertirían en estándares en la industria como *Sid Meier's Civilization* en 1991, que creó el género "4X" (eXplorar, eXpandir, eXplotar, eXterminar) dentro de los juegos de estrategia[2]. En la *Figura 1* observamos la carátula del sexto juego de la saga, lanzado en 2016, atestiguando la longevidad de la serie.

Figura 1 Caratula de Sid Meier's Civilization VI (2016)

2.1.3 Nacimiento de los RTS

En 1992 la empresa Virgin Games publica *Dune II*[3], que, si bien no es el primer RTS asentó las bases de todos los RTS posteriores, por esto es considerado el arquetipo del género. *Dune II* está basado en la película *Dune* de 1984, una adaptación de la novela del mismo nombre escrita por Frank Herbert[25]. En el juego, el jugador es el comandante militar de una casa noble del imperio, la casa *Atreides*, *Harkonnen* u *Ordos*. Estas casas han sido enviadas al planeta *Arrakis* por el emperador, que ofrece el gobierno del planeta a la casa que más cantidad de especia sea capaz de entregarle. La especia es una droga del universo de Dune que se encuentra solo en *Arrakis*.

Cada casa cuenta con sus propias unidades especiales o variaciones de unidades comunes a las tres. Por ejemplo, la casa Harkonnen tiene el tanque "Devastator" que tiene armadura y artillería pesada, la casa Atreides tiene el tanque "Sonic Tank", que utiliza un cañón sónico y la casa Ordos tiene el tanque "Deviator" que dispara un gas nervioso que cambia la lealtad de las unidades a la casa Ordos durante un período de tiempo. *Arrakis* es un planeta árido cuya superficie es prácticamente en su totalidad desértica y está habitado por los gigantescos gusanos de arena que merodean en los desiertos del planeta y son capaces de devorar incluso vehículos. Estas características limitan la construcción de edificios a las pocas zonas rocosas del planeta. Durante las partidas el objetivo es destruir la base enemiga. El recurso con el que cuenta el jugador son los créditos, que se obtienen de la extracción de especia y su transformación en créditos en una refinería. Los créditos se emplean tanto para la construcción de edificios como para el reclutamiento de unidades. El juego fue un éxito de ventas, alcanzando las 250.000 unidades vendidas en 1996, y debido a esto el juego además de contar con su versión para PC recibió en 1993 versiones para *Commodore Amiga* y la consola *Mega Drive* y más recientemente, en 2013 se lanzó una versión del juego para Android que cuenta con más de 100.000 descargas en la tienda de Google Play[25].

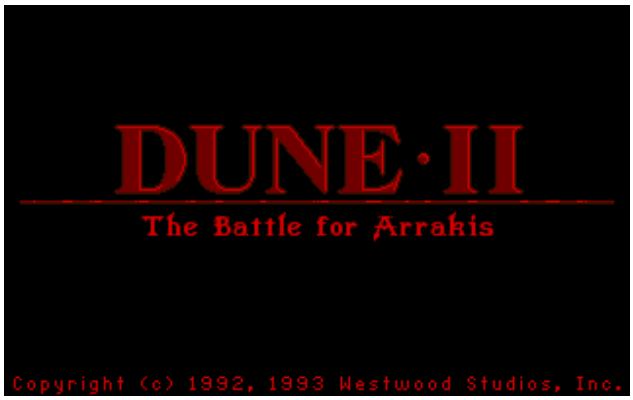


Figura 2 Pantalla de título de Dune II (1992)

Los videojuegos RTS enfatizan la acumulación de recursos mediante la recolección en minas de oro, canteras de piedra, bosques, etc. a través de unidades civiles, la construcción de una base con distintos edificios, teniendo cada uno su propia función, y la producción de unidades de combate. Su característica principal es el combate en tiempo real a diferencia de otros géneros de estrategia.

La gran popularidad de *Dune II* inspiró las hoy en día sagas legendarias de *Warcraft* (1994), *Command and Conquer* (1995), *Age of Empires* (1997) y *StarCraft* (1998)[3]. Este boom del género RTS provocó que se convirtiese en uno de los géneros de videojuegos más populares de la década de los 90 y del primer lustro del siglo XXI. El auge en la popularidad de otros géneros de videojuegos en la primera década del siglo los ha reducido a juegos de nicho con una base de jugadores pequeña pero leal que sigue jugando a estos juegos. Esta caída en popularidad explica por ejemplo que tras el lanzamiento del *Age of Empires III* en 2005 hayan transcurrido 16 años, hasta 2021 hasta el lanzamiento del *Age of Empires IV*, si bien Microsoft ha desarrollado otros títulos de estrategia como *Age of Empires Online* en 2011[11] que no tuvieron éxito.

Otro caso digno de mención es el de los RTS desarrollados por Blizzard Entertainment. En primer lugar, la saga de videojuegos de la cual nació el universo de fantasía del mismo nombre, *Warcraft*. Los RTS de *Warcraft* son una trilogía, *Warcraft: Orcs & Humans* (1994), *Warcraft II: Tides of Darkness* (1995) y *Warcraft III: Reign of Chaos* (2002) con su expansión *Warcraft III: The Frozen Throne* (2003) que culminaron en el MMORPG[1] *World of Warcraft* (2004) el cual contó con una grandísima popularidad gracias al éxito cosechado por los juegos que le precedieron[4]. La otra gran franquicia de RTS desarrollada por Blizzard Entertainment es *StarCraft* (1998) con su secuela *StarCraft II: Wings of Liberty* siendo publicada en 2010. A pesar de utilizar gráficos 2D cuando otros RTS de la competencia como *Age of Empires* ya utilizaban gráficos en 3D y de haber heredado muchos elementos del "gameplay" de sus predecesores *StarCraft* sentó precedentes en varias áreas.

[1] Juego de rol multijugador masivo online. Videojuego multijugador que se caracteriza por tener miles de jugadores en un mismo servidor.

Figura 3 Partida de StarCraft (1998)

En primer lugar, al igual que en la saga *Command & Conquer*, la historia del juego se presentaba al jugador a través de personajes profundos que recibieron vida con actores de voz cuando en la mayoría de RTS la historia se presentaba mediante textos largos que podían ser omitidos sin que el jugador perdiese realmente nada. En segundo lugar, *StarCraft* es uno de los videojuegos RTS con el multijugador más balanceado, significando que las características únicas de cada facción hacen única la jugabilidad según la facción utilizada, sin otorgar ventaja sobre las demás. Las facciones están diseñadas de forma asimétrica, es decir, cada una tiene edificios, unidades y habilidades diferentes en contraposición a otros juegos en los que las distintas facciones son más similares entre sí. En tercer lugar, la característica principal de la jugabilidad del *StarCraft* es la gestión y control de las unidades, que es un factor muy importante en el combate y es esencial para utilizar las habilidades especiales de las unidades de forma eficaz. Además, el juego cuenta con un editor de mapas que permite a los jugadores incluso modificar las reglas y objetivos del juego, habiendo surgido del editor de mapas del *StarCraft* el mod que se considera como predecesor de los videojuegos MOBA[2] que son un género de videojuegos en el que dos equipos de jugadores tienen como objetivo destruir la base del equipo rival ayudados de oleadas de súbditos, unidades de combate generadas automáticamente que siguen un camino predefinido hacia la base enemiga. Gracias a estos factores nació y se desarrolló una escena competitiva de *StarCraft* convirtiéndose en uno de los primeros E-sports[3], datando los primeros torneos profesionales del juego del año 2000 e inspirando muchas de las características de muchos RTS que vinieron tras él[5].

2.1.4 Evolución y subgéneros de los RTS

En la década de los 2000 surgieron RTS que trataron de evolucionar la fórmula de los RTS con distintas innovaciones. Así, por ejemplo, en el año 2000 se publica el videojuego *Metal Fatigue*, un RTS ambientado en el siglo XXIII en el cual la humanidad ha desarrollado la tecnología para viajar a velocidades superiores a la de la luz. La mayor innovación del juego se encuentra en sus mapas. En los RTS convencionales el mapa es una capa de terreno con alteraciones geográficas

[2] Multiplayer Online Battle Arena
[3] Competiciones profesionales de videojuegos multijugador

como cuerpos de agua, montañas, etc. pero siempre en la superficie del mundo del juego. En *Metal Fatigue*, el mapa se divide en tres niveles o capas que representan distintas partes del planeta. Estas capas son el subterráneo del planeta, la superficie del planeta y las capas superiores del planeta, en las que hay asteroides orbitando el planeta y se pueden utilizar como plataformas de construcción. Así pues, *Metal Fatigue* plantea al jugador el complejo desafío de expandirse, combatir y gestionar sus recursos en las tres capas del mapa durante la partida.[23]

Otra alteración a la fórmula original de los RTS es la introducida por los juegos *Warhammer 40,000: Dawn of War* (2004)[12], *Star Wars: Empire at War* (2006)[13] y la saga *Company of Heroes* (2006)[14]. Estos juegos reemplazan el tradicional sistema de recolección de recursos sustituyéndolo por uno de puntos de control que, al ser capturados por el jugador, son los que le otorgan recursos. De esta manera, los jugadores se enfrentan al desafío de tener que capturar y mantener suficientes puntos de control para no quedarse atrás en recursos frente a los enemigos. El sistema de puntos de control también añade nuevas formas de poder ganar la partida, además de la tradicional forma de destruir la base enemiga añadiendo en el mapa "puntos de victoria", que son puntos de control especiales que no producen recursos, pero deben ser controlados durante un periodo de tiempo para ganar.

Figura 4 Punto de recursos en el Company of Heroes (2006)

Finalmente tenemos los MOBA (Multiplayer Online Battle Arena), subgénero del RTS que combina elementos de videojuegos RTS con elementos de rol y de acción. En los MOBA, el jugador controla a un héroe con habilidades especiales y junto a un equipo compuesto por otros jugadores, se enfrentan a otro equipo de jugadores que también controlan otros héroes con el objetivo de destruir la base del equipo contrario. El predecesor de este género es el mod del *StarCraft* llamado *Aeon of Strife*. En el mod un equipo de 4 jugadores ayudados por unidades más débiles controladas por la IA debía enfrentarse a la IA con el objetivo de destruir la base de esta. *Aeon of Strife* sentó las bases de lo que se convertiría en los MOBA.

Figura 5 Partida de Aeon of Strife, predecesor del género MOBA

La siguiente evolución de los MOBA se produjo en el siguiente RTS de Blizzard Entertainment, *Warcraft III: Reign of Chaos*, que contaba con un editor de mapas más potente que el del *StarCraft*. En el *Warcraft* surgió el mod *Defense of the Ancients*, abreviado DOTA por sus jugadores, en el que gracias al sistema de héroes del juego en el que estos tenían habilidades especiales y podían ser equipados con objetos, se creó la base de la cual evolucionarán los MOBA, en DOTA el objetivo era destruir el "Anciano" de la base enemiga para ganar la partida mientras proteges el de tu base[15].

En 2009 se publicaron los primeros juegos independientes utilizando la fórmula desarrollada en DOTA. *Demigod* es un videojuego en el que los héroes eran semidioses compitiendo entre ellos por ascender a la divinidad que fracasó por ser publicado con solo 8 héroes jugables y unos servidores multijugador que con frecuencia fallaban. También en 2009 se publicó el videojuego *League of Legends* por Riot Games, que hoy en día es uno de los videojuegos más jugados del mundo y cuenta con los eventos de E-sports más grandes del mundo con cientos de millones de personas viéndolos cada año[15].

2.1.5 Castle Strike dentro del género RTS

Como videojuego RTS, *Castle Strike* tiene similitudes y diferencias con los juegos presentados en los apartados anteriores. Está fuertemente inspirado en *Warcraft* y en *Age of Empires* por lo que puede ser considerado un RTS clásico, con los objetivos de recolectar recursos, construir fuerzas y destruir la base enemiga. La estética y la interfaz del videojuego son más similares a *Warcraft*, mientras que mecánicamente se parece más a *Age*

of Empires. En la siguiente *Figura* se pueden observar las similitudes visuales entre *Castle Strike* y *Warcraft.*

Figura 6 Comparación entre Castle Strike y Warcraft

Por otra parte, cada unidad de *Castle Strike* es diferente, con sus puntos fuertes y débiles como las unidades en *Age of Empires*. En ambos juegos son necesarios una gran cantidad de campesinos para mantener una buena economía y la principal fuente de producción de comida son las granjas, aunque en *Age of Empires* deben trabajar en ellas los campesinos y en *Castle Strike* no.

2.2 Arquitecturas de videojuegos multijugador en tiempo real

En un videojuego multijugador es importante tener en cuenta los requerimientos a la hora de elegir una arquitectura puesto que no es lo mismo un juego por turnos, que en tiempo real, o un juego de estrategia que un FPS[4], donde en uno es tolerable que haya una latencia elevada mientras que en otro tiene que ser lo más reducida posible, así pues es necesario comparar distintas arquitecturas[16] para elegir la que mejor se ajuste a lo que necesita *Castle Strike*. Las métricas utilizadas para comparar las distintas arquitecturas son las siguientes:

- Latencia, referente a la capacidad de respuesta del videojuego, el retraso entre que el jugador emite un input y el servidor lo procesa.
- Escala o escalabilidad, como de fácil es aumentar el número de jugadores en una partida.
- Seguridad, como de fácil es de vulnerar y si pone en riesgo las máquinas de los usuarios.
- Coste ¿Se puede conseguir un coste razonable por usuario para que el juego sea rentable?
- Complejidad, cómo de compleja es la tecnología utilizada.
- Alcance, cuántas personas pueden jugar al juego simultáneamente.

[4] Género de videojuegos que simula el uso de armas de fuego desde una perspectiva de primera persona.

La arquitectura básica de un videojuego multijugador en tiempo real es la siguiente:

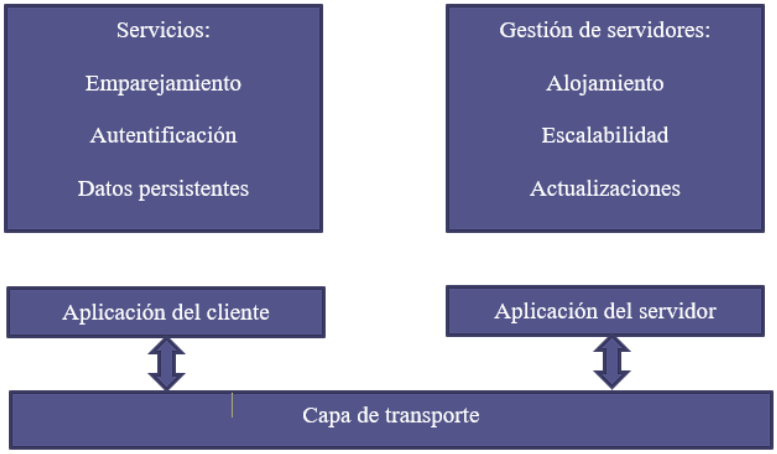


Figura 7 Arquitectura de un videojuego multijugador en tiempo real[16]

Los siguientes apartados[16] describen utilizando las métricas mencionadas las distintas arquitecturas para videojuegos multijugador que existen.

2.2.1 Multijugador Local

Todos los jugadores están juntos y utilizan la misma máquina para jugar. Al solo tener la aplicación del cliente es como un juego normal, pero procesando múltiples entradas de los jugadores que pueda haber. No tiene mayor dificultad que el desarrollo de un videojuego de un solo jugador.

- Latencia: Ideal, no hay gestión de red ✓✓
- Escala: Muy pequeña, típicamente, ≤4 jugadores ✗ ✗
- Seguridad: Riesgo mínimo ✓
- Coste: Ninguno, gratis ✓✓
- Complejidad: La arquitectura más simple ✓✓
- Alcance: Limitado a la localización física de la máquina ✗ ✗

2.2.2 Multijugador LAN

Con esta arquitectura los jugadores utilizan cada uno su propia máquina y se conectan entre ellos o se conectan a un servidor cerrado que no está conectado a internet. Esta arquitectura pues utiliza N aplicaciones de cliente conectadas todas entre sí o una aplicación de servidor a la cual se conectan todos los clientes.

- Latencia: Negligible ✓✓
- Escala: >100 jugadores utilizando un servidor ✓✓. Muy pequeña si los clientes se interconectan por la complejidad de sincronizarlos a todos ✗ ✗
- Seguridad: Riesgo mínimo ✓
- Coste: Ninguno, gratis ✓✓
- Complejidad: Simple ✓
- Alcance: Limitado a la localización física de la máquina ✗ ✗

2.2.3 Multijugador Peer-to-peer (P2P)

La arquitectura P2P es aquella en la que las máquinas de los jugadores están conectadas entre ellas y se encargan de procesar todos los datos necesarios para jugar sin conectarse a un servidor. Existen tres tipos de P2P, P2P directo, P2P con servidor-cliente y P2P con servidor-cliente a través de un relay.

En P2P directo los jugadores están conectados directamente todos entre ellos. Es la arquitectura más simple y directa, pero es también la que pone más carga en las máquinas de los clientes puesto que cada uno tiene que recibir N-1 inputs del resto de jugadores y sincronizarlos y debido a esto no permite una gran cantidad de jugadores.

En P2P con servidor-cliente uno de los jugadores actúa de servidor. El resto de los jugadores se conectan al jugador que actúa de servidor y es este el encargado de procesar y sincronizar todos los inputs. Esto significa que es la máquina del jugador que aloja la partida la que tiene que gestionar toda la carga. Por otra parte, en juegos competitivos el jugador que aloja la partida tiene ventaja respecto a los demás porque no tiene latencia al actuar de servidor. Un problema de este diseño es que, si el jugador que aloja la partida decide salir de esta o bien la partida termina, se debe de escoger a un nuevo jugador para actuar de servidor y se debe migrar a todos los jugadores a este nuevo servidor manteniendo la sincronización de la partida lo que es un proceso costoso de hacer y una muy mala experiencia de usuario. El P2P con relay funciona igual que el P2P con servidor-cliente, pero los jugadores se conectan al relay y este transmite los datos al jugador que aloja la partida, que los devuelve al resto de jugadores a través del relay.

	P2P Directo	P2P cliente-servidor	P2P con Relay
Latencia	Rápida si los jugadores están cerca ✓ Depende en gran medida de la conexión de los jugadores ✗ Sin ventaja de jugador-servidor ✓	Rápida si los jugadores están cerca ✓ Conexión estable ✓ Ventaja de jugador-servidor ✗	Latencia añadida con el relay ✗ Conexión estable ✓ Ventaja de jugador-servidor ✗
Escala	8-12 jugadores ✗	10-20 jugadores ✗	10-20 jugadores ✗
Seguridad	Cheats/Hacks posibles ✗ IPs de los usuarios expuestas ✗	Cheats/Hacks posibles ✗ IPs de los usuarios expuestas ✗	Cheats/Hacks posibles ✗ Solo es visible la IP del relay ✓
Coste	Barato, solo requiere emparejamiento ✓	Barato, solo requiere emparejamiento ✓	Moderado, requiere emparejamiento y el relay ✗
Complejidad	Es difícil sincronizar N jugadores ✗	Tecnología moderadamente compleja ✗ Complejidad de Migración de servidor ✗	Tecnología Compleja ✗ Complejidad de Migración de servidor ✗
Alcance	Limitado ✗	Limitado ✗	Mucho alcance (dependiente del relay)✗

2.2.4 Multijugador con servidor dedicado

Esta arquitectura utiliza servidores controlados por los administradores del juego de forma que los jugadores se conectan a la aplicación del servidor la cual recibe y sincroniza los inputs de los usuarios. Un servidor dedicado permite que muchos jugadores jueguen a la vez ya que es mucho más potente que los ordenadores de los usuarios, permite prevención de trampas y puede alcanzar a jugadores por todo el mundo. Finalmente, aunque es la arquitectura de mayor coste y más compleja son variables que se pueden controlar, se pueden reducir costes y se puede mejorar la tecnología para reducir su complejidad.

- Latencia: Rápida si los servidores están cerca ✓;
  Consistente entre todos los jugadores ✓
- Escala: Puede llegar a +100 jugadores ✓✓
- Seguridad: Prevención de trampas, autoridad del servidor ✓ ; IPs seguras ✓
- Coste: Alto, requiere alojar el servidor y los distintos servicios que debe proveer ✕
- Complejidad: Muy complejo, requiere una arquitectura completa ✕✕
- Alcance: Puede llegar a todo el mundo ✓✓

3. Análisis del problema

3.1 Game Design Document

El ciclo de vida del software comienza con la elicitación de requisitos y el diseño y en el desarrollo de videojuegos esto no es diferente. El GDD, como será referido a partir de ahora, es el documento donde se especifican los requisitos y el diseño del videojuego. En el GDD se incluyen todas las características que se desea que tenga el videojuego una vez finalizado. Sin embargo, estos requisitos pueden ser modificados o eliminados durante el desarrollo y el producto final puede no contener todo lo indicado en el GDD.

El GDD de *Castle Strike* comienza enunciando una descripción del videojuego.

"*Castle Strike* es un videojuego de estrategia 3D del género RTS para Microsoft Windows en el cual se tiene que reclutar un ejército para derrotar al resto de jugadores enemigos. El videojuego consistirá en un modo multijugador de 2 a 4 jugadores que lucharán por equipos o individualmente con el objetivo de destruir la base enemiga."

A continuación, se detalla el público objetivo y la clasificación PEGI[5] del videojuego, *Castle Strike* está orientado a personas a partir de los 12 años. La edad mínima está orientada según la clasificación PEGI del videojuego (PEGI 12) y su descripción es la siguiente: "Los videojuegos que muestran violencia de una naturaleza un poco más gráfica hacia los personajes de fantasía o violencia no realista hacia los personajes humanos entrarían en esta categoría de edad. Puede haber insinuaciones sexuales o posturas sexuales, mientras que cualquier lenguaje soez en esta

[5] Pan European Game Information. Proporciona clasificaciones por edad para videojuegos en Europa. Confirma que el juego es apropiado para jugadores de cierta edad.

categoría debe ser leve.”[17] *Castle Strike* encaja en esta descripción puesto que contiene violencia no realista hacia personajes humanos.

3.1.1 Directrices de jugabilidad

Este apartado explica brevemente como ocurrirán las partidas. Como en un RTS tradicional, *Castle Strike* pone el énfasis de su jugabilidad en la construcción de bases, recolección de recursos y combate con el objetivo de destruir la base del jugador enemigo. Al ser un videojuego multijugador no cuenta con un sistema de dificultad tradicional en el que el jugador se enfrenta a una IA más o menos poderosa, si no que la dificultad depende de la diferencia de habilidad entre los jugadores que se enfrenten entre sí.

3.1.2 Mecánicas del juego

Las mecánicas de un videojuego son todas aquellas herramientas de las que dispone el jugador para afectar a la partida. En *Castle Strike* se podrían dividir en las distintas unidades y edificios que puede utilizar el jugador, así como los recursos del mapa y las acciones que puede tomar el jugador durante el transcurso normal de la partida.

Figura 8 Unidades

Figura 9 Unidades especiales

En *Castle Strike* las unidades actúan individualmente como en juegos como *Age of Empires* o *Warcraft* y en contraposición a los sistemas de pelotones de juegos como *Company of Heroes* o *Warhammer 40,000: Dawn of War*, y se subdividen en tres categorías; unidad civil, el campesino, su función es la recolección de recursos y construcción de edificios y aunque pueden combatir son muy débiles. Unidades de combate básicas, el espadachín y el arquero conforman el grueso del ejército del jugador y si bien no destacan tienen un bajo coste de recursos. Unidades especiales, son el caballero y la catapulta. El caballero es la unidad de combate más poderosa del juego, tiene mucho ataque y velocidad de movimiento, pero es lenta y costosa de reclutar y la catapulta es una unidad de asedio, causa mucho daño, pero no es efectiva contra unidades porque sus proyectiles son fáciles de esquivar, además de ser vulnerable a los ataques de otras unidades por su baja velocidad de movimiento.

Figura 10 Edificios de Castle Strike

En el juego el jugador cuenta con estos cuatro edificios: el ayuntamiento es el edificio principal y el jugador solo puede poseer uno, puesto que perderlo significa perder la partida. El cuartel y el campo de tiro son los edificios militares, en ellos se producen las unidades de combate cuerpo a cuerpo y las unidades de combate a distancia respectivamente, y finalmente, el granero es el edificio de producción de comida del juego.

El jugador dispone de cuatro recursos diferentes que debe recolectar y que son necesarios para la construcción de edificios y el reclutamiento de unidades. Se pueden subdividir en dos grupos, recursos para la construcción y recursos para el reclutamiento. Los recursos de construcción son la piedra y la madera y los recursos de reclutamiento son la comida y el oro. La piedra, madera y oro se recolectan en el mapa de la partida. Distribuidos por el mapa existen canteras de piedra, depósitos de oro y bosques de los cuales los campesinos del jugador deben extraerlos y llevarlos al ayuntamiento del jugador y la comida se produce en el granero. Esto significa que el oro, la madera y la piedra son finitos, y una vez extraídos todos los que haya en el mapa no se puede obtener más cantidad, creando la necesidad en los jugadores de competir por los recursos del mapa.

3.1.3 Controles del Juego

El juego se controla con ratón y teclado. Los controles son los típicos de los juegos RTS, seleccionar unidades y darles órdenes de movimiento o de ataque y seleccionar los edificios para reclutar unidades en ellos.

Acción	Control
Selección de unidades	Click izquierdo con el ratón sobre la unidad
Selección múltiple de unidades	Mantener click izquierdo y arrastrar el ratón sobre las unidades
Movimiento	Click derecho sobre un punto del mapa
Orden de Ataque	Click derecho sobre una unidad o estructura enemiga
Orden de recolección de recursos	Con uno o más campesinos seleccionados, Click derecho sobre el recurso a recolectar
Orden de construcción	Con uno o más campesinos seleccionados, Click derecho sobre el edificio a construir
Movimiento de cámara	WASD
Movimiento rápido de cámara	Shift + WASD
Rotación de la cámara	Click central de ratón + Movimiento del ratón

Originalmente en el primer diseño del juego se plantearon más controles, como la creación de grupos de control, que son grupos de unidades cuya función es seleccionar muchas unidades fácilmente o también la posibilidad de establecer puntos de reunión en los edificios de reclutamiento para ordenar a las unidades recién reclutadas ir a un punto en específico del mapa, pero debido a la escala del juego finalmente se decidió no implementar estas características.

Típicamente el movimiento de la cámara en los juegos RTS se realiza desplazando el ratón hacia los bordes de la pantalla, pero hoy en día es común tener dos o tres monitores y jugar en ventana maximizada para alternar entre los monitores más cómodamente por lo que en muchos juegos que utilizan este método de desplazamiento es común que el cursor del ratón salga de la ventana del juego provocando clicks en otras ventanas o en el escritorio del ordenador y afectando negativamente a la experiencia por lo que en *Castle Strike* el movimiento de cámara se realiza con las teclas.

3.1.4 Estética del juego e Interfaz de Usuario

Los modelos 3D que se utilizan en el videojuego tienen una estética que se conoce como low-poly. El Low-poly es una técnica de modelado 3D que utiliza un número muy bajo de polígonos, de ahí su nombre. Se utiliza ampliamente para juegos o aplicaciones en tiempo real porque al utilizar modelos con poco detalle permite un procesamiento gráfico mucho más rápido.[18] A esta gran ventaja del Low-poly se le suma el hecho de que permite crear escenarios muy diversos y bonitos a un coste, tanto computacional como económico, mucho menor que otras técnicas de modelado 3D. Un ejemplo de esto sería el videojuego *Astroneer*, mostrado en la *Figura 11*.

Figura 11 Imagen del videojuego low-poly Astroneer

En *Castle Strike* los modelos utilizados para las unidades y edificios han sido extraídos de un paquete de assets de la *Unity Asset Store* llamado *Toony Tiny RTS set*.

Figura 12 Vista del Toony Tiny RTS set

El paquete cuenta con mucho contenido y, además, los modelos 3D están completamente animados e incluyen algunos efectos visuales como fuego para cuando se daña a los edificios.

Los menús del juego son simples y para navegar entre ellos el usuario debe de hacer click en los distintos botones disponibles. Como pantalla principal que actúa de punto de entrada de la aplicación está el menú principal desde el cual el usuario puede navegar al resto de menús que son el de opciones y el de jugar o bien salir de la aplicación. En el menú de opciones el jugador puede ajustar el nivel de volumen del juego y en el menú de jugar puede crear partidas nuevas o unirse a partidas ya existentes. A continuación, se presenta el diagrama de navegación de la aplicación, así como los mock-ups de los menús del juego.

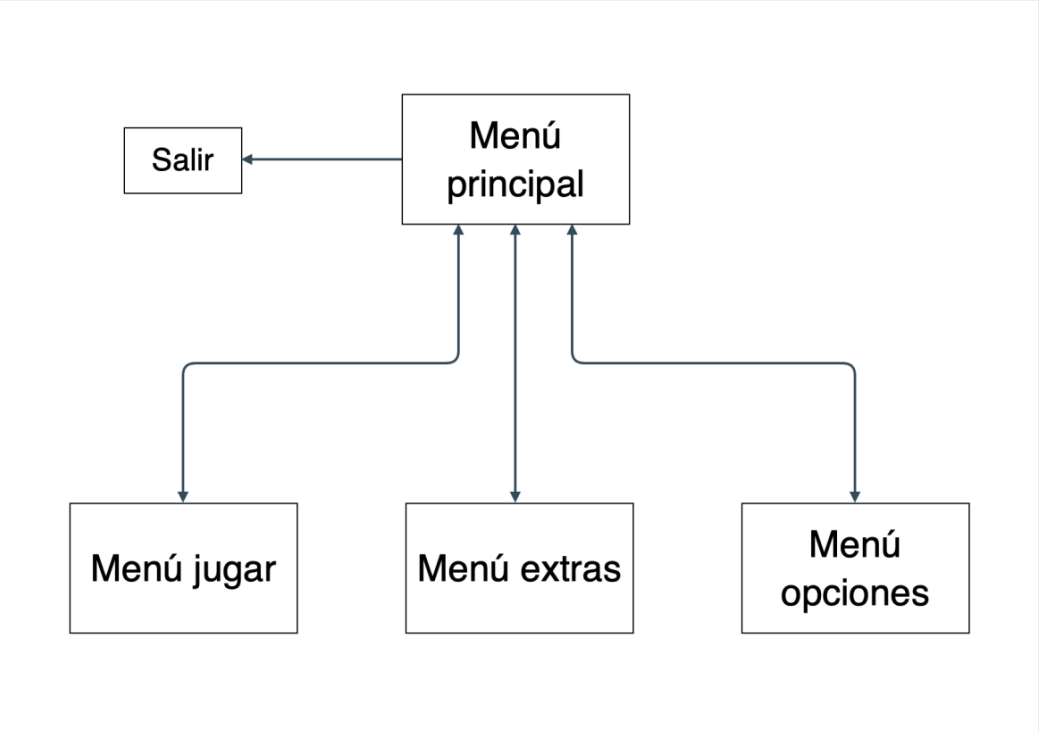


Figura 13 Diagrama de navegación de la aplicación

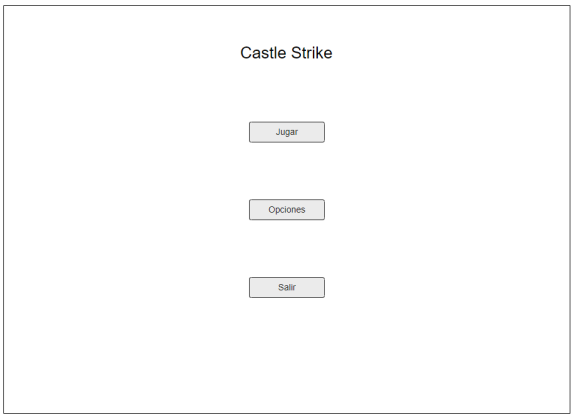


Figura 15 Menú Principal

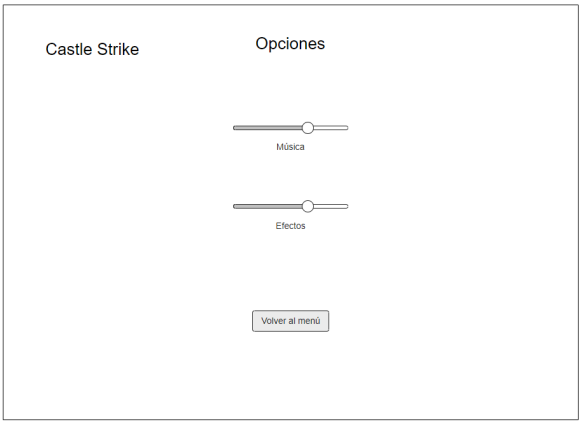


Figura 14 Menú de Opciones

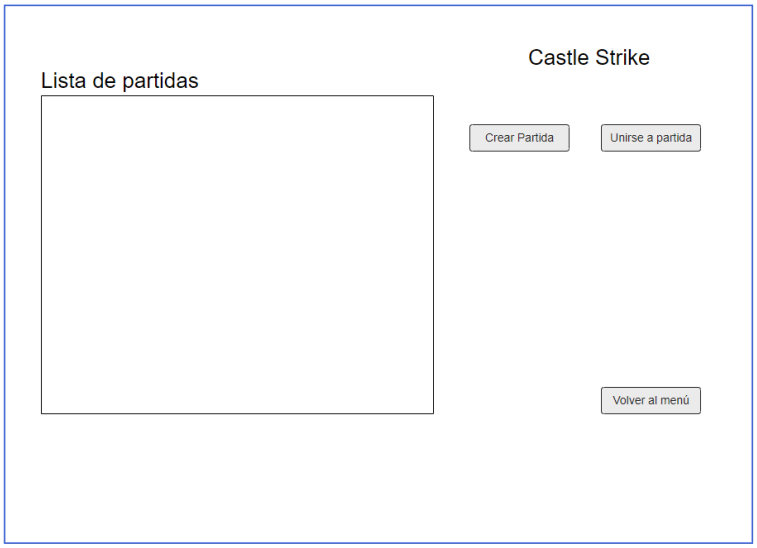


Figura 16 Menú de selección de partida

El diseño de la interfaz dentro de la partida está inspirado en las del *Age of Empires* y *Warcraft III* y es el diseño típico de interfaz de los videojuegos RTS con un panel en la parte superior de la pantalla en el que se indican los recursos actuales del jugador y con botones para acceder a las opciones del juego, y otro panel en la parte inferior de la pantalla en el que se muestra la selección actual del jugador, acciones que pueden realizarse con lo que esté seleccionado y el mini mapa de haber uno. La interfaz de partida de *Castle Strike* se ha desarrollado utilizando el pack de contenido *Classic RPG GUI* de la Unity Asset Store. En la siguiente ilustración se mostrará el diseño final de la interfaz del juego.

Figura 17 Interfaz de la partida

3.2 Análisis de la Seguridad

En el apartado 2.2 se han explicado las distintas arquitecturas existentes para videojuegos multijugador en tiempo real y entre las características explicadas se encuentra la seguridad. El framework *Photon PUN 2*, es la solución de desarrollo multijugador que se ha utilizado en *Castle Strike* y es una solución de P2P directo con relay, de forma que no tiene cliente-servidor. Por lo tanto, las IPs de los usuarios están protegidas. Sin embargo, no se puede incluir ninguna forma de prevención de trampas porque todo el código se ejecuta en las máquinas de los usuarios y esto permite que se puedan utilizar programas como *Cheat Engine*, que permite acceder a la memoria de aplicaciones en ejecución y modificarla, para hacer trampas. La aplicación tampoco cuenta con sistemas de autentificación de usuarios, ya que no es necesario para jugar, ni almacena ningún tipo de información personal de los jugadores por lo que no existe la posibilidad de que estos datos puedan ser robados.

3.3 Análisis de la Propiedad Intelectual

A la hora de desarrollar videojuegos en Unity utilizando contenido de la *Unity Asset Store* es muy importante conocer el EULA[19] de Unity que dicta cómo se deben utilizar los contenidos licenciados en la *Unity Asset Store* para evitar incurrir en problemas legales. En concreto el apartado relevante en esta situación es el 2. *END-USER'S Rights and Obligations* ya que es el que define cómo se pueden utilizar los assets adquiridos.

Existe la posibilidad de que la licencia del asset no se rija por el EULA de Unity y que tenga sus propios términos de licencia. En esta situación Unity distingue entre "Non-Restricted Assets" a los que se aplica el EULA de Unity y "Restricted Assets" que son este segundo grupo de assets con sus propios términos. Los assets utilizados en *Castle Strike* son todos "Non-Restricted Assets" por lo que se rigen por el EULA de Unity.

Así pues, la licencia estándar de Unity permite incorporar el asset junto a contenido original no obtenido de la *Unity Asset Store* a una aplicación electrónica con un propósito, características y funciones que vayan más allá de mostrar, distribuir o usar los assets como parte integral de la aplicación de forma que el asset no comprenda una parte sustancial del producto desarrollado.

Concluyendo, una vez adquiridos los assets se puede hacer con ellos lo que se desee dentro de juegos y/o aplicaciones desarrolladas con ellos, incluido monetizar estas aplicaciones y lo que no está permitido es duplicar y revender los assets como propios.

3.4 Presupuesto

Es posible desarrollar juegos en Unity que utilicen solamente assets gratuitos si existen assets que cubran todas las necesidades del proyecto. En este trabajo no ha sido posible porque requería unos assets muy específicos y el tener que desarrollarlos habría hecho imposible completar el proyecto por el trabajo extra que habría requerido, véase diseño, modelado y animación 3D y 2D para los modelos de unidades y edificios y las interfaces del juego, así como diseño de sonido y audio en general del juego.

Debido a estas limitaciones fue necesaria la adquisición de los paquetes de assets desglosados a continuación:

- Toony Tiny RTS Set. Precio: 15.13€
- Classic RPG GUI. Precio: 5.40€
- Strategy Game Icons. Precio: 10.79€
- HUMBLE SOFTWARE BUNDLE: UNITY FANTASY GAMES & GAME DEV ASSETS. Precio: 21.37€

Total: 52,69€

Nótese que el pack HUMBLE SOFTWARE BUNDLE: UNITY FANTASY GAMES & GAME DEV ASSETS contenía una gran cantidad de assets de Unity y no todos han sido utilizados en el proyecto.

Para completar el presupuesto es necesario también calcular el coste estimado de la programación. Habiendo consultado a dos personas que trabajan en Pendulo Studios y en Digital Sun, dos estudios de videojuegos españoles, el salario de un programador junior es de alrededor de 20.000€ anuales, aproximadamente 9,60€/h. Junto con la estimación de 340h de la propuesta de este proyecto el resultado es de 3.264€.

Por lo tanto, el coste total aproximado del proyecto es de 3.316,69€.

Finalmente, aunque se ha utilizado la licencia gratuita de PUN conviene estudiar las distintas licencias que ofrece y los precios de estas para poder valorar el coste de comercializar el juego. Las distintas licencias de PUN ofrecen un aumento en el número de usuarios concurrentes que pueden conectarse al juego y ofrecen la capacidad de externalizar la gestión de servidores y la infraestructura, además de tener la capacidad de escalar automáticamente según la demanda de los usuarios. La licencia gratuita permite un máximo de 20 usuarios concurrentes. La licencia PLUS ofrece 100 usuarios concurrentes por un precio de 95$ mensual y existen planes hasta los 2000 usuarios concurrentes con un precio de 370$ mensual o también existe la opción de contratar *Photon PREMIUM Cloud* con capacidad de escalar automáticamente según demanda y un precio variable costando 0,29$ por usuario al mes.

4. Diseño de la solución

4.1 Tecnología Utilizada

En este apartado se compararán el motor de juego utilizado y el framework de desarrollo multijugador escogido con otras alternativas y se explicará porqué se han escogido frente a las alternativas.

4.1.1 Motor de juego

Para la elección de un motor de juego se consideraron dos alternativas, ambas factibles, que habrían permitido desarrollar el proyecto: Unity y Unreal Engine.

Unity es considerado el motor de juego por excelencia para desarrolladores novatos porque empezar a usarlo es muy sencillo, cuenta con una gran cantidad de guías y materiales de aprendizaje y una gran comunidad a la que consultar para resolver dudas y problemas.

El desarrollo con Unity está enfocado al uso de GameObjects[6], que componen las escenas del juego. Las escenas son los distintos menús y niveles que componen el juego y contienen gameObjects que son todos los componentes con los que los usuarios pueden interactuar.

Un gameObject puede ser cualquier cosa dentro de una escena: desde un botón a una unidad o un script que se ejecuta en la aplicación. La función que cada gameObject desempeña varía según los componentes que se le asignen como se muestra en las imágenes a continuación:

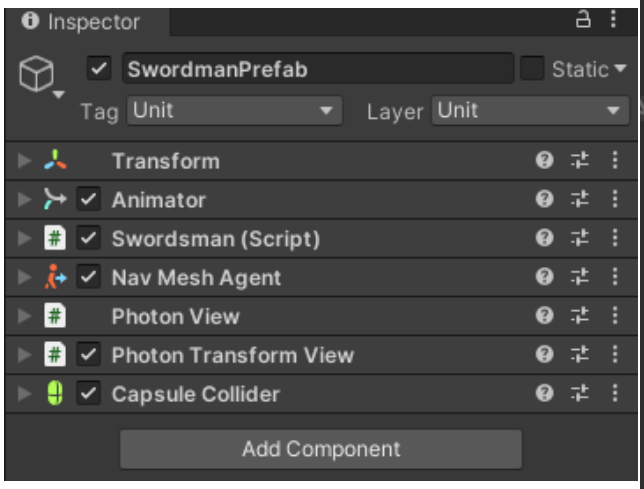


Figura 18 Componentes de un gameObject

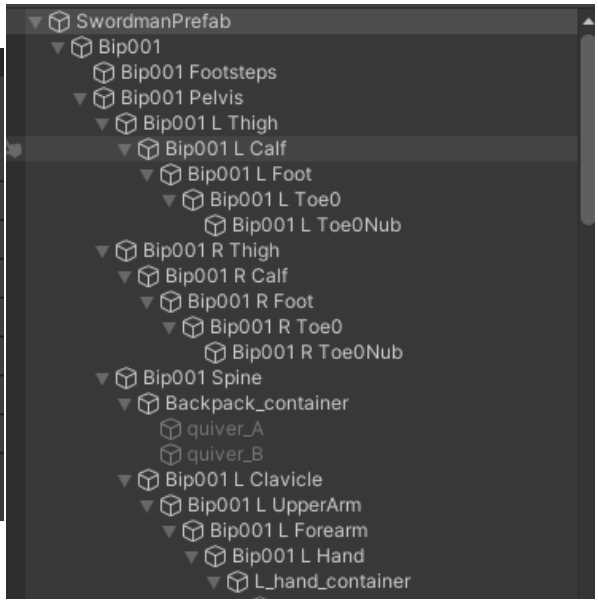


Figura 19 gameObjects en la jerarquía

[6] Unidad básica para construir las escenas del juego.

En la *Figura 18* Componentes de un gameObject podemos observar un gameObject con sus componentes. Hay un componente común a todos los gameObjects que es el componente *Transform*, que contiene la posición en la escena del gameObject, su rotación y tamaño. El resto de componentes son opcionales y se añaden para dotar al gameObject de una funcionalidad específica, por ejemplo, el componente *Animator* contiene una máquina de estados finita en el que cada estado es una de las animaciones de la unidad y las transiciones entre estas animaciones. La *Figura 19* nos muestra la jerarquía de escena de Unity, en la jerarquía se muestran todos los gameObjects en la escena y las relaciones entre ellos. Tanto por motivos organizativos como funcionales un gameObject puede tener gameObjects hijo que están vinculados a este y pueden contener parte de la funcionalidad del gameObject o bien estar organizados para conseguir una jerarquía más limpia y ordenada, facilitando el desarrollo.

La programación en Unity se realiza en C# y, a diferencia del desarrollo tradicional o en otros motores, está basada en scripts. Los scripts se añaden a los gameObjects como componentes y definen el funcionamiento del gameObject. A partir de la Unity Scripting API podemos modificar otros componentes del gameObject o afectar otros gameObjects y de esta manera crear la experiencia de juego deseada.

En contraposición a Unity, Unreal Engine es conocido por su gran fidelidad gráfica permitiendo crear juegos con gráficos muy realistas empleando tecnologías como el ray-tracing[7] o el hiperrealista motor de físicas de Unreal Engine, así como su propio framework multijugador con una arquitectura cliente-servidor. La base del editor de Unreal Engine es similar al de Unity. Los juegos también se componen de escenas que tienen objetos con componentes. La programación en Unreal Engine se realiza en C++.

Por motivos personales originalmente el proyecto iba a realizarse en Unreal Engine con el objetivo de aprender C++ durante el desarrollo. Sin embargo, tras valorar y comparar ambos motores de juego se escogió Unity. Los motivos fueron: la experiencia previa de haber desarrollado ya tres proyectos en Unity, dos de ellos en 2D, el tener conocimientos de C# y el completo desconocimiento de Unreal Engine y de la programación en C++.

4.1.2 Framework Multijugador

Como se menciona en el apartado anterior, Unity cuenta con gran cantidad de recursos para apoyar a los desarrolladores y a la hora de elegir un framework de desarrollo multijugador esto no es diferente, ya que existe un informe de Unity Technologies que analiza y compara 5 soluciones distintas y es el documento que se ha utilizado para informar la decisión.[24] En el documento se analizan los siguientes frameworks: MLAPI, DarkRift 2, Photon PUN, Photon Quantum v2 y Mirror. En el apartado 3.2 se ha explicado que la solución utilizada en el proyecto es Photon PUN.

MLAPI (Mid-level API) es una librería de código abierto que ofrece muchas características como RPCs[8], gestión de escenas y variables de red. No es complicada de utilizar, aunque ofrece pocas guías y tutoriales para usuarios que están comenzado a utilizarla. Es una solución con muy buen rendimiento, permitiendo latencias de 50-100ms, ideales para juegos rápidos como

[7] Algoritmo de síntesis de imágenes que simula el comportamiento físico de la luz para alcanzar un mayor realismo.
[8] En computación distribuida, es una técnica utilizada para ejecutar código en otra máquina distinta a la que realiza la llamada.

FPS. No soporta procesamiento multihilo, por lo que puede presentar problemas con la escalabilidad y finalmente no ofrece ningún servicio de alojamiento de servidores.

DarkRift 2 es una solución de bajo nivel que no ofrece características de medio y alto nivel como RPCs o variables de red ni ofrece UDP, utilizando tan solo TCP para la comunicación. Incluye procesamiento multihilo por defecto, haciéndola la solución más escalable de todas y permite latencias de 50ms o menos. No ofrece tampoco ninguna solución dedicada de alojamiento y tiene un coste de 100$ para desbloquear todas las características que incluye.

Photon PUN es una solución P2P que ofrece características como RPCs y un sistema de emparejamiento de jugadores muy simple de utilizar. De todas las soluciones es la más fácil de utilizar con multitud de ejemplos y guías prácticas disponibles en su documentación y en la web. La latencia en juegos desarrollados con PUN es elevada, entre 150-200ms por lo que no es apto para juegos rápidos y al ser P2P tiene una escalabilidad muy limitada, pero esto no supone un problema para juegos pequeños. PUN ofrece sus propios servidores de forma gratuita hasta 20 usuarios concurrentes, lo que la hace ideal para proyectos pequeños, ya sean personales o académicos.

Photon Quantum cuesta 1000$ al mes por lo tanto fue descartado inmediatamente.

Mirror es un fork de HLAPI de UNet, la antigua solución de Networking de Unity ya obsoleta y reemplazada por MLAPI, que ha sido mantenida y ampliada por la comunidad. Es más estable que su predecesora y tiene una comunidad muy activa capaz de resolver cualquier duda que pueda surgir. Incluye características como RPCs pero tiene problemas de gestión de memoria que afectan negativamente al rendimiento y a la escalabilidad de la solución. Es gratuita y de código abierto y no ofrece tampoco ningún servicio de alojamiento.

Tras estas consideraciones se escogió PUN porque al ser un juego pequeño la poca escalabilidad del P2P no es un problema. En una partida de un RTS es muy raro que haya más de 8 jugadores, para el tipo de juego desarrollado no es necesaria una latencia muy reducida y finalmente es una solución muy fácil de utilizar con todo el transporte siendo transparente al programador. Y el alojamiento gratuito en los servidores de PUN la convierten en la solución ideal para este proyecto.

4.2 Diseño de la Solución

En la siguiente imagen se muestra el diagrama de clases que contiene las clases referentes al código utilizado para jugar una partida de *Castle Strike*. No es todo el código del juego, sin embargo, debido a la estructura de los videojuegos desarrollados con Unity, el resto de clases, relacionadas con menús y cambio de escenas, no interactúan entre sí y por ello no se ha considerado relevante incluirlas en el diagrama.

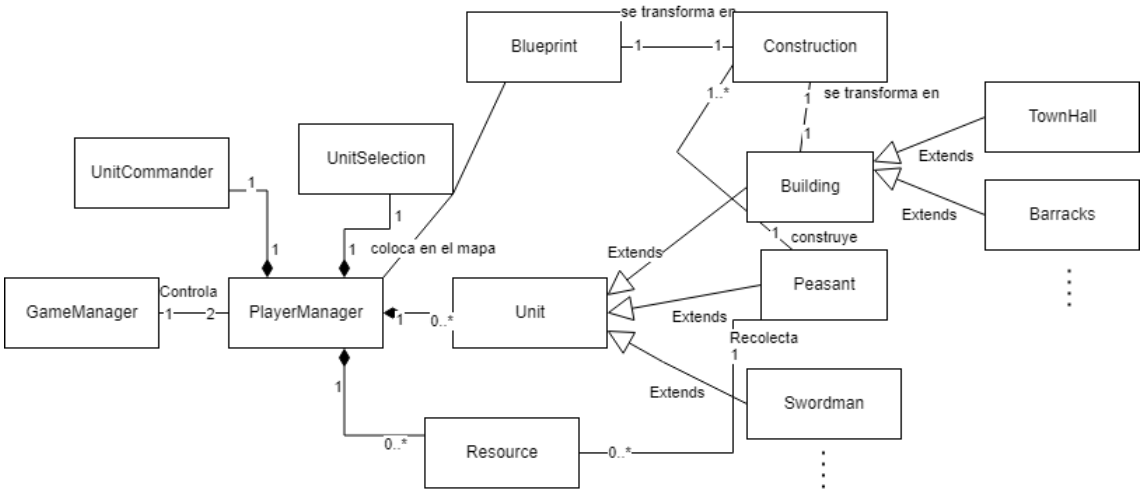


Figura 20 Diagrama de clases

Nótese que de la clase Unit heredan todas las unidades, pero por simplicidad se han omitido en el diagrama, con la clase Building ocurre lo mismo para los edificios. Todos los scripts del diagrama de clases pertenecen a un gameObject del mismo nombre, a excepción de UnitCommander y de UnitSelection que pertenecen a PlayerManager, ya que para poder ejecutar código en Unity este debe estar anclado a un gameObject que exista en la escena en ejecución. La clase GameManager, como su nombre indica se encarga de gestionar la partida y contiene referencias a los jugadores para poder iniciar la partida cuando todos hayan cargado, generar las estructuras y unidades iniciales de estos y terminar la partida una vez un jugador derrota al otro. La clase PlayerManager contiene todo lo que pertenece al jugador y controla la interfaz de usuario en la partida. Tiene referencias a todas sus unidades, sus edificios y sus recursos, a través de las clases UnitSelection y UnitCommander manipula las unidades para seleccionarlas y darles órdenes respectivamente. También se controla la construcción de edificios a través de PlayerManager. Si el jugador tiene a uno o más Peasant seleccionados puede construir edificios mediante el siguiente proceso:

Al hacer click en el botón apropiado de la interfaz del jugador, mostrada en la *Figura 17*, el jugador generará una Blueprint que se utiliza para determinar la ubicación de un edificio en el mapa. Esta Blueprint se desplaza con el ratón a una posición válida y al confirmar se transforma en una Construction que son los cimientos de un edificio y al terminar se transforma en la Building correspondiente.

La clase Unit representa todo aquello que el jugador puede seleccionar y controlar, por este motivo las unidades y los edificios heredan de Unit, la clase contiene todos los atributos y métodos comunes a todas estas clases como son su salud, ataque, velocidad de ataque o velocidad de movimiento si bien la clase Building sobrescribe parte de esta funcionalidad común ya que un edificio ni ataca ni se mueve. Los scriptable objects son un tipo especial de clase de Unity utilizado como contenedor de datos cuya función es reducir el uso de memoria del proyecto evitando tener más de una copia de la misma variable. Esto hace que cada vez que se instancia una nueva unidad, en lugar de que esta tenga sus propias variables para sus atributos utilice la información contenida en su scriptable object, que es idéntica para todas las unidades del mismo tipo. La clase Peasant, Swordman y las demás clases de unidades extienden la funcionalidad de Unit según las tareas específicas de cada unidad. Así la clase Peasant puede construir edificios y recolectar recursos y contiene una inteligencia artificial simple para la recolección de recursos de forma que la realice automáticamente sin dirección del jugador hasta

que reciba otra orden diferente, por otra parte la clase Swordman tiene la capacidad de detectar unidades enemigas cercanas para atacarlas automáticamente sin que el jugador tenga que estar pendiente todo el tiempo de sus unidades o de lo contrario podrían matarlas sin que llegaran siquiera a combatir. Cada clase que hereda de Building amplía la funcionalidad de esta con las características de cada edificio, su scriptable object con sus atributos, y las unidades que cada edificio recluta o en el caso de granero producir comida.

Y finalmente, Resource es una clase simple que se asigna a cada gameObject que sea un nodo de recursos, árbol, depósito de oro y depósito de piedra, para identificarlo con su tipo correspondiente para que cuando un Peasant lo recolecta tenga la información de qué recurso está recogiendo.

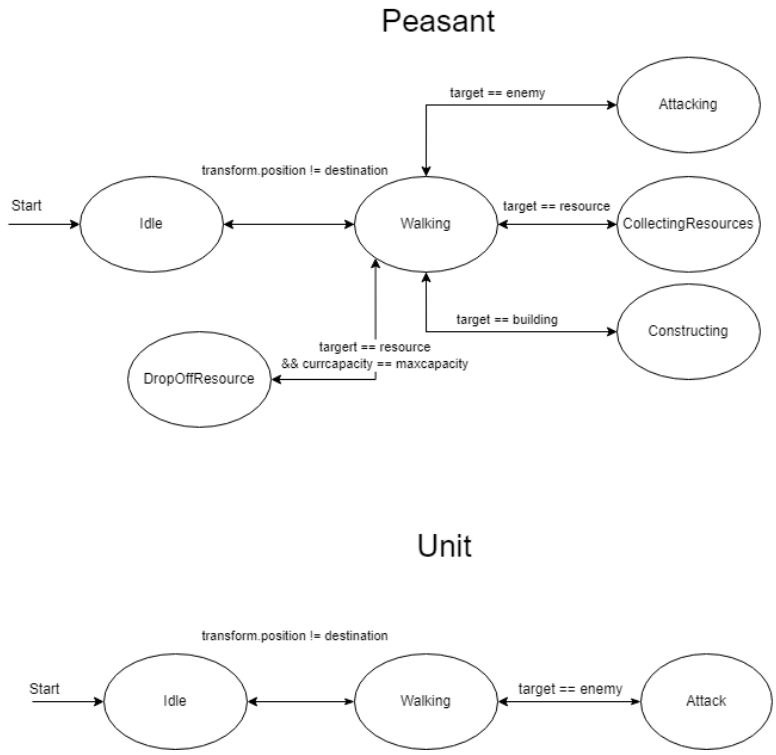


Figura 21 Diagrama de estado del campesino y de las unidades

En la *Figura 21* se muestra el diagrama de estado de las unidades del juego. El campesino al realizar más tareas que las unidades de combate tiene más estados. Por otra parte, la simplicidad de las unidades ha facilitado mucho su implementación ya que al tener el mismo comportamiento ha permitido que en su mayoría utilicen el mismo código. Si observamos la *Figura 22* comprobamos que los diagramas de estados coinciden con los Animator de los campesinos y de las unidades, nótese que la *Figura* solo muestra el Animator del campesino. En el lateral izquierdo de la imagen se encuentran los parámetros del Animator que son más o menos equivalentes a las condiciones de los cambios de estado del diagrama. Son valores booleanos que se controlan mediante script y activan las transiciones entre los estados del Animator, determinando la animación que se debe reproducir.

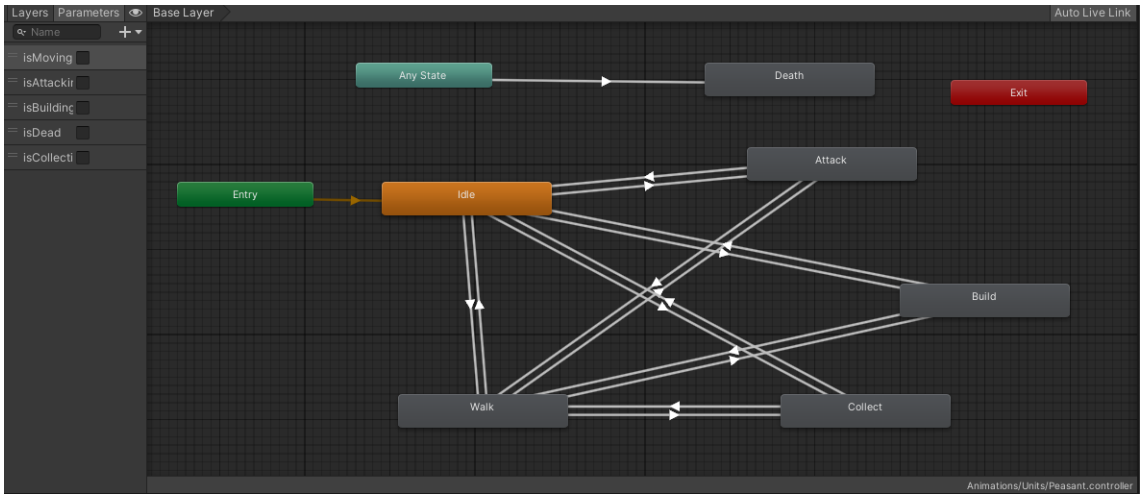


Figura 22 Animator del campesino

Para implementar la funcionalidad multijugador, todos los gameObjects a sincronizar deben contener componentes del plugin de PUN que se añade al proyecto de Unity. El componente fundamental para la sincronización del que dependen los demás es el Photon View que se utiliza para identificar un gameObject a través de la red, asignándole una ID y configurando cómo se sincroniza. A partir de la Photon View operan el resto de los componentes de PUN, como el Photon Transform View, que sincroniza la posición, rotación y escala del gameObject y el Photon Animator View, que sincroniza las animaciones del gameObject. Existen más componentes como el Photon Teams Manager, para gestionar equipos de jugadores en caso haber, pero estos no se han sido utilizado.

No toda la sincronización se puede realizar mediante los componentes auxiliares de la Photon View, como podrían ser acciones que deben ejecutarse simultáneamente en más de una máquina. Esta funcionalidad se realiza mediante llamadas RPC que permiten ejecutar un método especifico a través de la red para que ocurra en los ordenadores de los otros jugadores o con la función de PUN *OnPhotonSerializeView*, que permite sincronizar variables a través de la red. Un ejemplo de esto es el ataque entre las unidades que debe realizarse en las máquinas de todos los jugadores para mantener la salud de las unidades sincronizadas. Una desventaja de *OnPhotonSerializeView* es que es unidireccional, lo que significa que solo puede utilizarlo para enviar datos el dueño de la Photon View en la que se ejecuta y el resto de clientes solo puede recibir dichos. Por este motivo es útil para sincronizar cambios que un jugador produce localmente como la cantidad de recursos que posee. Y la principal ventaja es que si no hay nada que sincronizar PUN no hace la llamada a la función, ahorrando tiempo de ejecución y ancho de banda. Para el resto de las acciones que se requiere sincronizar, que son acciones realizadas por un jugador a otro, se debe utilizar una llamada RPC. Retomando el ejemplo anterior con el ataque de las unidades, al ser una llamada RPC, se ejecuta la función de ataque en ambas máquinas provocando el mismo resultado en los dos ordenadores. La clase Unit utiliza tres métodos RPC para el ataque de las unidades, *SetTargetRPC*, *AttackRPC* y *ClearTargetRPC*. Cuando el jugador ordena a una unidad atacar primero se realiza una llamada a *SetTargetRPC* que sincroniza el objetivo de la unidad en el ordenador del dueño de la unidad y en el del otro jugador, tras esto si el objetivo está fuera de alcance de la unidad que ha recibido la orden primero se acerca hasta estar al alcance, una vez le alcanza se realiza la llamada a *AttackRPC* que provoca que la unidad ataque a su objetivo de forma sincronizada entre ambos jugadores y

finalmente si la unidad destruye su objetivo *ClearTargetRPC* limpia el objetivo de la unidad para que no siga atacando a la nada. El proceso se muestra en la siguiente *Figura*:

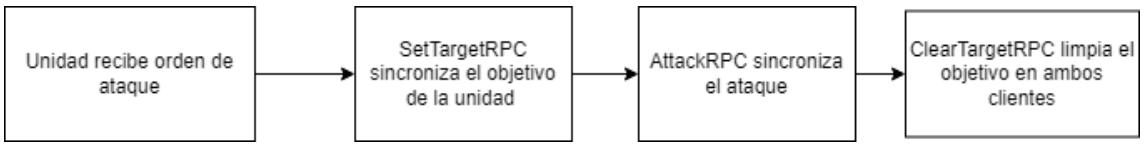


Figura 23 Proceso de ataque de las unidades

El movimiento de las unidades es otra característica vital de un videojuego RTS. En el proyecto se ha implementado utilizando el sistema de navegación de Unity[20], que está formado por los siguientes componentes:

- NavMesh (Navigation Mesh): Estructura de datos que describe las superficies caminables del mundo del juego y permite encontrar un camino entre una posición y otra. Se construye automáticamente a partir de la geometría del nivel.
- NavMesh Agent: Componente de los gameObjects que utiliza la NavMesh para desplazarse e incorpora la capacidad para evitar a otros agentes y obstáculos estáticos o en movimiento.
- Off-Mesh Link: Componente de los gameObjects que representa caminos que no pueden ser representados en la NavMesh, como una valla que se puede saltar.
- NavMesh Obstacle: Componente de los gameObjects que describe objetos que los agentes deben tratar como obstáculos y evitar al desplazarse.

El área caminable del mapa se construye probando las zonas donde un agente puede estar, y se definen a partir de los parámetros que toma un agente:

- Radio: distancia mínima que puede haber entre el agente y las paredes del mundo del juego o entre otros agentes
- Altura: La altura del agente, cuánto espacio vertical necesita.
- Altura de paso: Altura máxima que puede recorrer un agente entre zonas discontinuas verticales, como por ejemplo la altura máxima de un escalón que es capaz de subir o bajar el agente.
- Pendiente máxima: Inclinación máxima que el agente es capaz de subir

Combinando estos parámetros junto con la geometría del nivel, Unity construye el NavMesh. Puede existir más de un agente distinto y que al tener parámetros distintos tengan áreas caminables distintas. En *Castle Strike* el mapa es completamente plano, por lo tanto, no hay ninguna zona por la que las unidades no puedan desplazarse. En la siguiente figura observamos el NavMesh del nivel.

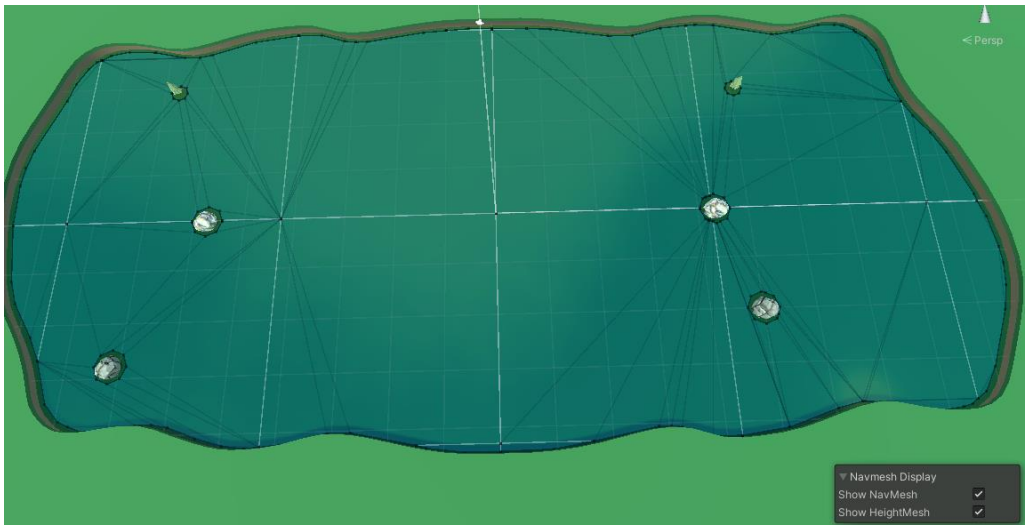


Figura 24 NavMesh del mapa de juego

Una vez configurada la NavMesh y los agentes estos tienen que recorrer los caminos que se les indique. En la *Figura 24* se pueden observar unas líneas negras que recorren la NavMesh y la dividen en polígonos. Estos representan un área dentro de la cual no existe ninguna obstrucción. Además, cada polígono también se guarda información sobre sus vecinos. Para encontrar un camino entre dos posiciones primero hay que seleccionar el polígono en el que se encuentran. Después, se busca desde la posición inicial pasando por todos los polígonos colindantes hasta llegar al polígono en el que se encuentra la posición final. Recorrer los polígonos visitados nos da el camino desde la posición inicial hasta el destino, Unity utiliza el algoritmo A* para calcular los caminos[21]. A* es un algoritmo de búsqueda en grafos que trata de encontrar el camino de menor coste entre la posición de origen y la de destino[22]. Dado que este trabajo no pertenece a la rama de computación y no es posible acceder a la implementación que hace Unity del algoritmo, no se va a explicar en detalle.

Durante la partida la superficie caminable del mapa va cambiando conforme los jugadores construyen edificios, los NavMesh Obstacle que contienen los edificios y puntos de recursos son los responsables de realizar estas modificaciones conforme son instanciados y destruidos los objetos que las contienen. De esta forma los agentes son capaces de modificar el camino que tienen que recorrer cada vez que cambia la NavMesh. En la *Figura 24* se puede ver en los puntos de recursos que hay en el mapa que no están dentro de la NavMesh. El área que cubre el obstáculo se configura en el editor de Unity con los parámetros del NavMesh Obstacle, como se muestra en la *Figura 25*. El NavMesh Agent cuenta también con sus propios parámetros de configuración que determinan su velocidad, velocidad angular, aceleración, distancia de frenado y el radio de detección de obstáculos entre otros como se muestra en la *Figura 26*.

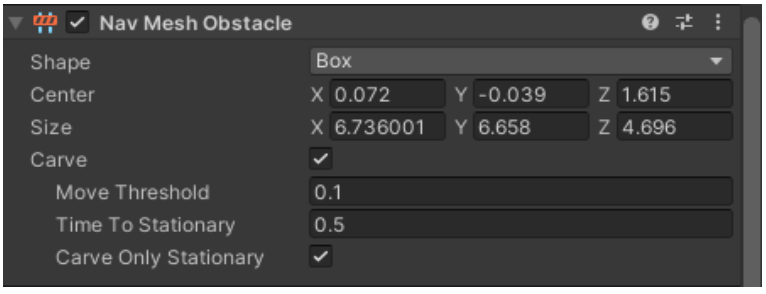


Figura 25 Parámetros de un NavMesh Obstacle

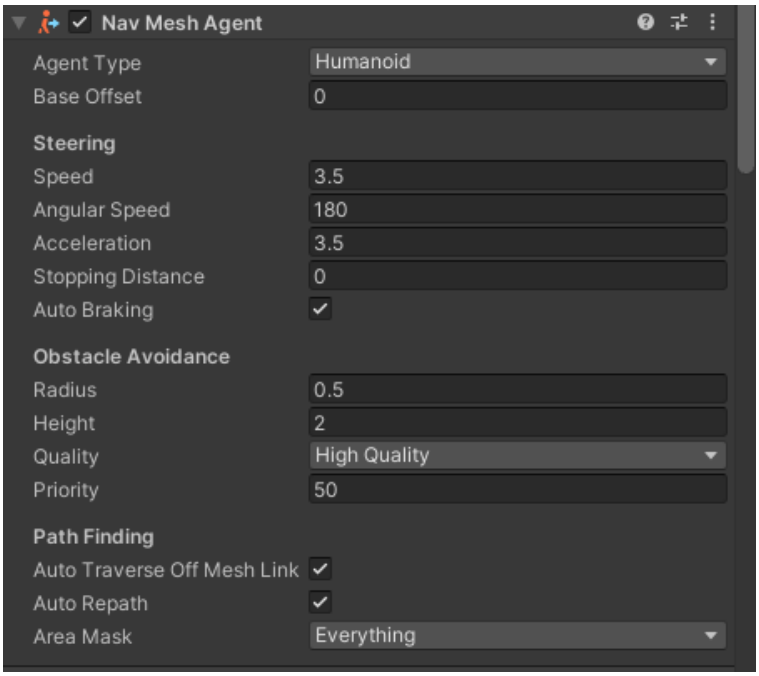


Figura 26 Parámetros de un NavMesh Agent

5. Desarrollo de la solución

5.1 Preparación del proyecto

El primer paso en el desarrollo es la instalación de Unity y la creación del proyecto. En la instalación de Unity en primer lugar se instala el software Unity Hub que se utiliza para gestionar los proyectos de Unity y las instalaciones de distintas versiones del editor. Para el proyecto se ha utilizado la versión de Unity 2020.3.11f1 LTS porque era la última versión LTS disponible cuando se empezó el proyecto. Como podemos ver en la *Figura 27* Unity cuenta con diversas plantillas para iniciar un proyecto con contenido básico a partir del cual comenzar el desarrollo, pero para *Castle Strike* escogemos un proyecto 3D vacío puesto que no sirve ninguna de las plantillas. También hay que vincular el proyecto con un repositorio de Github para tener control de versiones y no perder el proyecto.

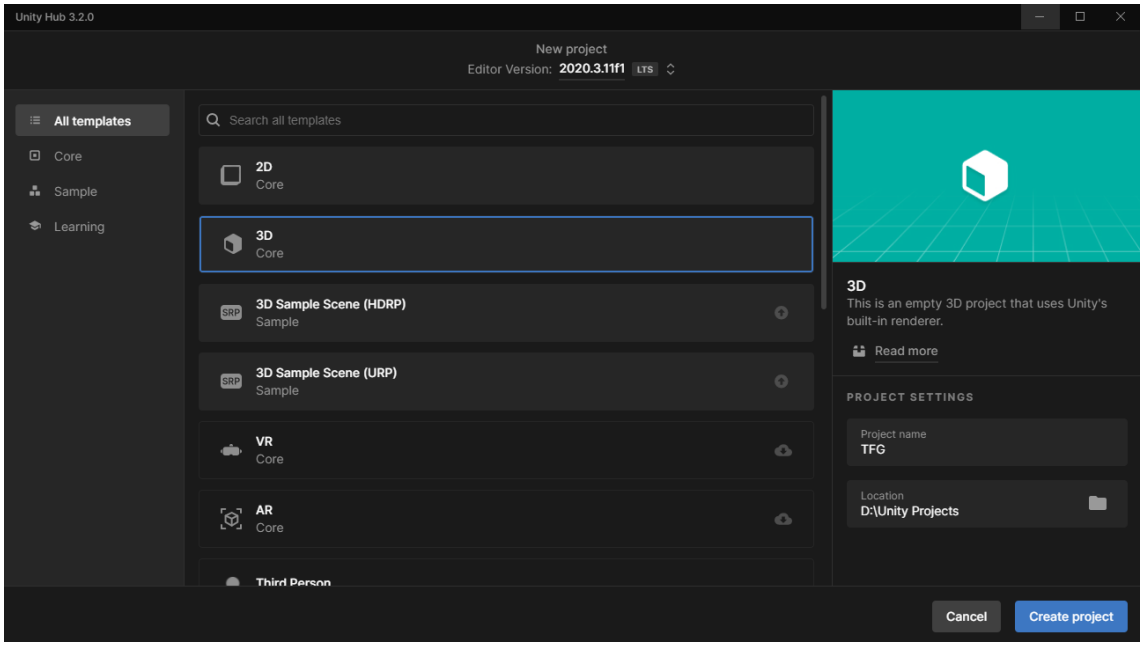


Figura 27 Creación de proyecto en Unity

Una vez realizada la configuración inicial puede comenzar el desarrollo.

5.2 Desarrollo

Debido a la extensión del proyecto no se comentará todo el desarrollo, sino que se hará hincapié en los aspectos más complicados del proceso y las dificultades encontradas.

5.2.1 Instanciación de objetos con PUN

En videojuegos offline para instanciar nuevos objetos en la escena se utiliza la función *GameObject.Instantiate(GameObject obj)*. Sin embargo, con PUN es necesario utilizar *PhotonNetwork.Instantiate(String path)*, donde path es la dirección del objeto a instanciar

dentro de la carpeta Resources del proyecto, por ejemplo, *Resources/Prefabs/Buildings/Townhall*. Esto crea la necesidad de, o bien guardar una referencia por objeto que necesitemos instanciar o de obtener la dirección en tiempo de ejecución cada vez que queramos instanciar algo. Como alternativa se ha aprovechado la capacidad de serialización de Unity y se han creado las clases *NetworkedPrefab* y *PrefabManager*. *NetworkedPrefab* es un objeto que contiene una referencia a un gameObject y su dirección en el directorio del proyecto y *PrefabManager* es una clase que implementa el patrón Singleton para poder acceder a su instancia desde cualquier otra clase, y guarda una lista de *NetworkedPrefab* y un método *NetworkInstantiate* que recibe una referencia a un objeto a instanciar y envuelve a *PhotonNetwork.Instantiate*. Nótese la anotación *[System.Serializable]* que indica a Unity que debe serializar y guardar los objetos de esta clase. El método *PrefabPathModified* acorta la dirección para que empiece tras resources porque es a partir de dónde pide la dirección PUN.

```
[System.Serializable]
public class NetworkedPrefab
{
    public GameObject Prefab;
    public string Path;

    public NetworkedPrefab(GameObject obj, string path)
    {
        Prefab = obj;
        Path = PrefabPathModified(path);
    }

    private string PrefabPathModified( string path )
    {
        int extensionLength = System.IO.Path.GetExtension(path).Length;
        int additionalLength = 10;
        int startIndex = path.ToLower().IndexOf("resources");

        if (startIndex == -1)
            return string.Empty;
        else
            return path.Substring(startIndex + additionalLength, path.Length -
(additionalLength + startIndex + extensionLength));
    }
}
```

Figura 28 Clase NetworkedPrefab

PrefabManager contiene también una función *PopulateNetworkPrefabs* que se encarga de llenar la lista de *NetworkedPrefab* y, al estar serializada, Unity actualiza el gameObject que contiene el código de *PrefabManager* y, por lo tanto, cada vez que se modifiquen los prefabs[9] se debe de ejecutar este código al menos una vez para actualizar la lista. La anotación *[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]* hace que la función se llame automáticamente al ejecutar el juego y la región *#if UNITY_EDITOR* es necesaria porque la clase *AssetDatabase* solo está disponible en el editor, por lo tanto de no incluir la región al intentar hacer una build del juego esta fallaría por contener código del editor.

[9] GameObject prefabricado.

```
public class PrefabManager : MonoBehaviour
{
    public static PrefabManager instance;

    [SerializeField]
    private List<NetworkedPrefab> _networkedPrefabs = new
List<NetworkedPrefab>();

    private void Awake()
    {
        if (instance != null && instance != this)
            gameObject.SetActive(false);
        else
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
    }

    public static GameObject NetworkInstantiate(GameObject obj, Vector3 position,
Quaternion rotation)
    {
        foreach (NetworkedPrefab networkedPrefab in instance._networkedPrefabs)
        {
            if (networkedPrefab.Prefab == obj)
            {
                GameObject result =
PhotonNetwork.Instantiate(networkedPrefab.Path, position, rotation);
                return result;
            }
        }
        return null;
    }

    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]
    private static void PopulateNetworkPrefabs()
    {
#if UNITY_EDITOR
        instance._networkedPrefabs.Clear();
        GameObject[] results = Resources.LoadAll<GameObject>("");
        for (int i = 0; i < results.Length; i++)
        {
            if (results[i].GetComponent<PhotonView>() != null)
            {
                string path = AssetDatabase.GetAssetPath(results[i]);
                instance._networkedPrefabs.Add(new NetworkedPrefab(results[i],
path));
            }
        }
#endif
    }
```

Figura 29 PrefabManager

Estas dos clases nos permiten evitar todos los problemas que provoca llamar la función *PhotonNetwork.Instantiate* directamente.

5.2.2 Unity Tags

Unity permite añadir un tag a los gameObjects para identificarlos y categorizarlos. A la hora de programar son muy útiles porque permite diferenciar las interacciones entre gameObjects según el tag que tengan. Sin embargo, un único tag es muy limitante en cuanto a posibilidades, sobre todo si se necesita subcategorizar los gameobjects, como en un inventario de objetos o en el caso de *Castle Strike* para subcategorizar las unidades y los edificios de forma que, por ejemplo, el cuartel tenga el tag de Building y el tag de Barracks. Para solucionar este problema se ha desarrollado un componente personalizado, *CustomTag*, que permite añadir a los gameObjects múltiples tags. Para acceder a los tags de Unity y poder manipular el componente directamente desde el editor también se ha creado *CustomTagEditor* que contiene código de editor que se encarga de mostrar *CustomTag* como un componente y le proporciona los tags guardados en Unity.

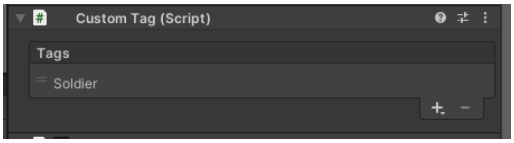


Figura 30 Vista del editor de Custom Tag

```
public class CustomTag : MonoBehaviour
{
    [SerializeField]
    private List<string> tags = new List<string>();

    public bool HasTag(string tag)
    {
        return tags.Contains(tag);
    }

    public IEnumerable<string> GetTags()
    {
        return tags;
    }

    public void Rename(int index, string tagName)
    {
        tags[index] = tagName;
    }

    public string GetAtIndex(int index)
    {
        return tags[index];
    }

    public int Count
    {
        get { return tags.Count; }
    }
}
```

Figura 31 Clase CustomTag

```
[CustomEditor(typeof(CustomTag))]
public class CustomTagEditor : Editor
{
    private string[] unityTags;
    SerializedProperty tagsProp;
    private ReorderableList list;

    private void OnEnable()
    {
        unityTags = InternalEditorUtility.tags;
        tagsProp = serializedObject.FindProperty("tags");
        list = new ReorderableList(serializedObject, tagsProp, true, true, true,
true);
        list.drawHeaderCallback += DrawHeader;
        list.drawElementCallback += DrawElement;
        list.onAddDropdownCallback += OnAddDropdown;
    }
```

Figura 32 Clase CustomTagEditor

Gracias al nuevo componente podemos subcategorizar los prefabs del proyecto como más conveniente sea. Un uso del componente es en PlayerManager a la hora de mostrar los iconos de unidades seleccionadas en la interfaz. La llamada a *SetBigIcon* que se muestra a continuación recibe el *CustomTag* de la unidad seleccionada y devuelve el icono que hay que mostrar.

```
GameObject bigFrame =
Instantiate(SetBigIcon(units[0].GetComponent<CustomTag>().GetAtIndex(0)),
                              unitFrame.transform);

private GameObject SetBigIcon (string tag)
    {
        return tag switch
        {
            "Peasant" => peasantBig,
            "Soldier" => soldierBig,
            "Archer" => archerBig,
            "Cavalry" => cavalryBig,
            "Catapult" => catapultBig,
            "Townhall" => townHallBig,
            "Farm" => farmBig,
            "ArcherRange" => archerRangeBig,
            "Barracks" => barracksBig,
            _ => peasantBig,
        };
    }
```

Figura 33 Ejemplo de uso de CustomTag

Es importante destacar que tanto la implementación de *PrefabManager* como la de *CustomTag* son reutilizables para cualquier proyecto de Unity que utilice PUN o necesite que los gameObjects contengan más de un tag.

5.2.3 NavMesh y NavAgents

Una dificultad encontrada durante el desarrollo del proyecto fue enviar las unidades a los edificios, ya fuera un campesino enviado a construir un edificio o un soldado con la orden de atacar un edificio enemigo. Como se ha explicado en el apartado 4.2, el NavMesh Obstacle crea un hueco en la NavMesh, y esto provocaba que las unidades que tenían que acercarse a los

edificios no pudieran porque tenían que desplazarse a una posición fuera de la NavMesh. Para solucionar este problema se decidió que las unidades se desplazaran a un punto en el perímetro del edificio situado a distancia suficiente como para estar en el interior de la NavMesh. En los GameObjects de los edificios se añadieron a su alrededor otros Gameobjects vacíos [10] y mediante la función *OffsetDestination* se determina cuál de los objetos en el perímetro del edificio es el más cercano a la unidad. En la siguiente Figura se muestra el código relevante.

```
        float closestDistanceSqr = Mathf.Infinity;
        Vector3 currentPosition = transform.position;
        Vector3 closestDistancePosition = Vector3.zero;

        foreach (GameObject destination in destinationPositions)
        {
            Vector3 directionToTarget = destination.transform.position -
currentPosition;
            float dSqrToTarget = directionToTarget.sqrMagnitude;
            if (dSqrToTarget < closestDistanceSqr)
            {
                closestDistanceSqr = dSqrToTarget;
                closestDistancePosition = destination.transform.position;
            }
        }

        destination = closestDistancePosition;
```

Figura 34 Código de OffsetDestination

5.2.4 Dificultades del desarrollo multijugador online

Aunque la documentación de PUN es muy detallada y a nivel básico es muy sencillo crear una escena simple con unos pocos gameObjects sincronizados entre ellos, la dificultad de desarrollar un videojuego de cierto tamaño crece exponencialmente y la dificultad de utilizar PUN en este proyecto no ha sido poca.

La gran cantidad de conceptos nuevos que se han tenido que aprender, y comprender, para poder trabajar con PUN han ralentizado y dificultado el desarrollo. La complejidad de probar un juego multijugador teniendo que ejecutar varias instancias del programa simultáneamente en un solo ordenador y sin poder acceder al inspector de los gameObjects ni a la jerarquía de escena para ver su estado han hecho también muy difícil encontrar fallos en el programa.

Al principio del desarrollo cuando el juego llevaba unos pocos minutos en ejecución la posición de las unidades se desincronizaba. Por ejemplo, en el jugador uno se desincronizaba las unidades del jugador dos y viceversa. Esto ocurría por un detalle muy simple de solucionar pero que no está explicado en la documentación de PUN. Un gameObject sincronizado con PUN existe en ambos ordenadores simultáneamente, y el gameObject tiene una serie de componentes, entre ellos el script que le otorga su comportamiento. Unity ejecuta localmente en cada ordenador el script cuando solo debe ejecutarse localmente en el ordenador del dueño del gameObject. La solución es comprobar al principio de los métodos de Unity que se ejecutan en cada frame, *Update*, *FixedUpdate*, *OnTriggerEnter*, etc, si el jugador local es el dueño del GameObject y, si no lo es, terminar la ejecución del método con un *return* tal y como se

[10] Gameobject que solo contiene el componente transform

muestra en la siguiente figura. Desactivar el script de los GameObjects que no pertenecen a un jugador localmente no es una opción porque entonces las llamadas RPC y la función *OnPhotonSerializeView* dejarían de funcionar.

```
private void OnTriggerEnter(Collider other)
    {
        if (!photonView.IsMine)
        {
            return;
        }
```

Figura 35 Comprobación de photonView.IsMine

No todo el trabajo de implementar PUN en el videojuego ha sido complejo como se muestra a continuación. La clase *NetworkManager* implementa el patrón Singleton y es la encargada de conectarse y desconectarse a PUN y contiene la lógica para crear y unirse a partidas, y junto con la clase *GameMenu* es la encargada de permitir a los jugadores crear partidas, unirse a ellas o abandonarlas. Entre otras funcionalidades *GameMenu* muestra la lista de los jugadores que se encuentran en la partida y la actualiza automáticamente cada vez que entra o sale un jugador.

```
public class NetworkManager : MonoBehaviourPunCallbacks
{
    public static NetworkManager instance;

    void Awake()
    {

        if (instance != null && instance != this)
            Destroy(gameObject);
        else
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
    }

    void Start()
    {
        PhotonNetwork.ConnectUsingSettings();
    }

    private void OnDestroy()
    {
        PhotonNetwork.Disconnect();
    }


    public void CreateRoom(string roomName)
    {
        PhotonNetwork.CreateRoom(roomName, new Photon.Realtime.RoomOptions {
MaxPlayers = 2});
    }

    public void JoinRoom (string roomName)
    {
        PhotonNetwork.JoinRoom(roomName);
    }
```

```
    [PunRPC]
    public void ChangeScene (string sceneName)
    {
        PhotonNetwork.LoadLevel(sceneName);
    }}
```

Figura 36 NetworkManager

```
public void OnCreateRoomButton (TMP_InputField roomNameInput)
    {
        if (playerName.text == "")
        {
            alertText.text = "Player Name cannot be empty";
            return;
        }
        if (roomName.text == "")
        {
            alertText.text = "Room Name cannot be empty";
            return;
        }
        alertText.text = "";
        NetworkManager.instance.CreateRoom(roomNameInput.text);
    }

    public void OnJoinRoomButton (TMP_InputField roomNameInput)
    {
        if (playerName.text == "")
        {
            alertText.text = "Player Name cannot be empty";
            return;
        }
        if (roomName.text == "")
        {
            alertText.text = "Room Name cannot be empty";
            return;
        }
        alertText.text = "";
        NetworkManager.instance.JoinRoom(roomNameInput.text);

    }

  public override void OnCreateRoomFailed(short returnCode, string message)
    {
        base.OnCreateRoomFailed(returnCode, message);
        alertText.text = message;
    }

    public override void OnPlayerLeftRoom(Photon.Realtime.Player otherPlayer)
    {
        UpdateLobbyUI();
    }

    [PunRPC]
    public void UpdateLobbyUI()
    {
        playerListText.text = "";

        foreach(Photon.Realtime.Player player in PhotonNetwork.PlayerList)
        {
            playerListText.text += player.NickName + "\n";
        }

        if (PhotonNetwork.IsMasterClient) startGameButton.interactable = true;
```

```
        else startGameButton.interactable = false;
    }

    public void OnLeaveLobbyButton()
    {
        PhotonNetwork.LeaveRoom();
        SetScreen(mainScreen);
    }
```

Figura 37 Parte de la clase GameMenu

En el apartado 4.2 se ha explicado el uso de llamadas RPC para sincronizar el ataque de las unidades. Para llegar a la implementación final de la mecánica se tuvo que iterar cuatro veces realizando implementaciones diferentes de la misma hasta diseñar una solución que funciona el 100% del tiempo. En las siguientes figuras se muestra la primera implementación del ataque y la versión final. La primera implementación hacía que la unidad objetivo se causase daño a sí misma, y en ocasiones fallaba sin motivo aparente. La implementación final es mucho más simple y con un código mucho más fácil de entender, la unidad que recibe la orden de atacar es la que realiza el ataque. La implementación definitiva es posible por el método *SetTargetRPC*, que sincroniza el objetivo de la unidad a través de la red, esto no era posible tampoco al realizar la primera implementación porque aún no entendía como hacerlo correctamente.

```
  public void Attack()
{
    {
        photonView.RPC("ReceiveDamage", RpcTarget.OthersBuffered,
            attackDamage, target.gameObject.GetPhotonView().ViewID);
        Debug.Log("Target viewID is: " +
target.gameObject.GetPhotonView().ViewID);
    }

    [PunRPC]
    public void ReceiveDamage(float damageReceived, int damagedUnitId)
    {
        //if (target.gameObject.GetPhotonView().ViewID == damagedUnitId) {
        target = PhotonView.Find(damagedUnitId).gameObject.transform;
        target.gameObject.GetComponent<Unit>().health -= damageReceived;
        Debug.Log("I am receiving damage");
        // }
        Debug.Log("ReceiveDamage is being called in ViewID: " +
photonView.ViewID);
    }      Debug.Log("ReceiveDamage is being called in ViewID: " + photonView.ViewID);

  }
```

Figura 38 Primera implementación del ataque

```
public void Attack()
    {
        photonView.RPC("AttackRPC", RpcTarget.All);
    }

    [PunRPC]
    public void AttackRPC ()
    {
        if (target != null)
        {
            Debug.Log("AttackRPC executing");
            transform.LookAt(target);
            if (target.CompareTag("Unit") || target.CompareTag("Peasant"))
target.GetComponent<Unit>().Health -= unitStats.attackDamage;
            else if (target.CompareTag("Construction"))
target.GetComponent<Construction>().health -= (int)unitStats.attackDamage;
            else if (target.CompareTag("Building"))
target.GetComponent<Building>().health -= unitStats.attackDamage;
        }

    }
```

Figura 39 Implementación final del ataque

Otras dificultades derivadas del desarrollo se han dado por desconocimiento del desarrollo multijugador online, errores causados por situaciones que no se han considerado o se han pasado por alto. Ejemplos de este tipo de errores son no limitar la partida al número de jugadores máximos o no bloquear el acceso a una partida tras empezarla, provocando que se pueda unir un jugador a una partida empezada. Este error es muy fácil de solucionar, tan solo hay que cambiar las propiedades IsVisible e IsOpen de la partida al iniciarla.

```
public void OnStartGameButton()
    {
        PhotonNetwork.CurrentRoom.IsOpen = false;
        PhotonNetwork.CurrentRoom.IsVisible = false;
        NetworkManager.instance.photonView.RPC("ChangeScene", RpcTarget.All,
"Game");
    }
```

Figura 40 Propiedades IsOpen e IsVisible

PUN es sencillo a nivel básico pero la dificultad de trabajar con él aumenta exponencialmente junto con los requisitos del proyecto.

6. Pruebas

El juego ha sido probado por 10 jugadores que han jugado entre ellos y contra mí. Las partidas que no he jugado las he observado desde la perspectiva de ambos jugadores comprobando que el funcionamiento del juego era el esperado y viendo cómo se desenvolvían en la partida. Tras finalizar las sesiones de juego los jugadores han respondido una encuesta que evaluaba su satisfacción en varios apartados del juego. También se ha recabado información de la duración de las sesiones de juego mediante el uso de Unity Analytics.

6.1 Observación de las partidas

Observar las partidas de los jugadores tenía dos objetivos:

- Comprobar el funcionamiento del juego y la aparición de bugs no detectados durante las pruebas en solitario durante el desarrollo.
- Observar si la información que proporciona el juego a los jugadores es suficiente para que fuesen capaz de jugar sin más indicaciones que una explicación de los controles.

Además, también permitió recibir valoraciones en directo del juego mientras los jugadores lo probaban por primera vez.

La prueba cumplió sus objetivos y se encontraron varios bugs entre ellos:

- Si se le ordenaba construir a un campesino mientras cargaba recursos, se quedaba parado sin hacer nada.
- Tras construir un edificio los campesinos dejaban de moverse y las siguientes tareas que se les mandase las realizaban quedándose estáticos desde donde estaban.
- Se podía construir edificios fuera del mapa.
- El pivote sobre el que rota la cámara está desplazado respecto a esta, causando una rotación extraña.

Por otra parte, al observar las partidas de los jugadores se extrajeron las siguientes conclusiones:

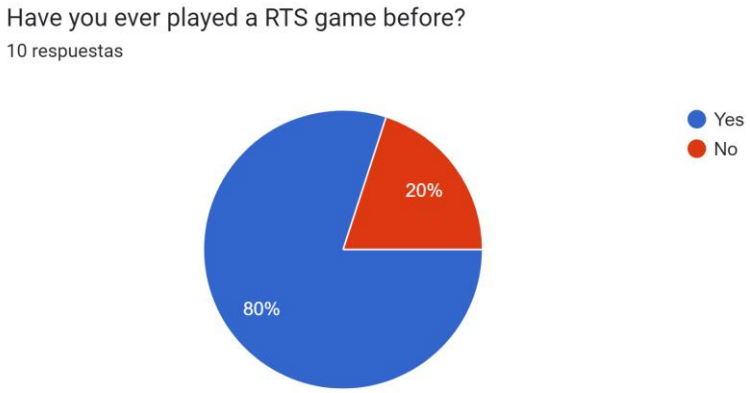
- Características comunes en los RTS como alertas al ser atacado o avisos de campesinos que no están haciendo nada son necesarias para la experiencia de juego.
- La interfaz no muestra suficiente información sobre los edificios y las unidades.
- Se requieren más valoraciones de jugadores para ser capaz de equilibrar las estadísticas de las unidades correctamente. Actualmente algunas tienen demasiada ventaja sobre otras.
- Los jugadores se divirtieron.

Como se va a ver más adelante algunas de estas valoraciones coinciden con las respuestas a la encuesta.

6.2 Análisis de la encuesta y Unity Analytics

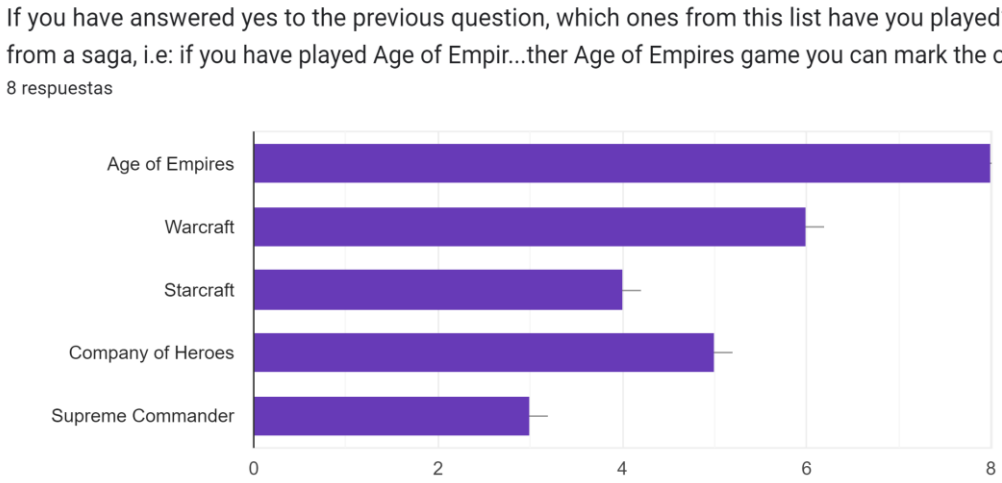
La encuesta está formada por 11 preguntas que se van a desglosar y analizar a continuación:

Primera Pregunta: ¿Habías jugado a un juego RTS antes?



De los 10 jugadores solamente 2 era la primera vez que probaban un RTS, tener tanto jugadores con experiencia como sin experiencia ha permitido conocer las expectativas de ambos grupos de jugadores y saber qué esperaba cada uno.

Segunda Pregunta: Si has respondido que sí a la pregunta anterior ¿Has jugado a algún juego de esta lista?



De los 8 jugadores que respondieron que sí a la primera pregunta el 100% ha jugado al *Age of Empires*, seguido de *Warcraft* con un 75% y de *Company of Heroes* con un 62,5%, esta pregunta es importante para las respuestas de la siguiente ya que están relacionadas.

Tercera Pregunta: ¿Te ha recordado *Castle Strike* a alguno de los juegos mencionados en la pregunta anterior? ¿Cuáles? ¿Cuáles son sus similaridades?

Has Castle Strike reminded you of any of the games mentioned on the previous question? Which one(s)? And what are their similarities

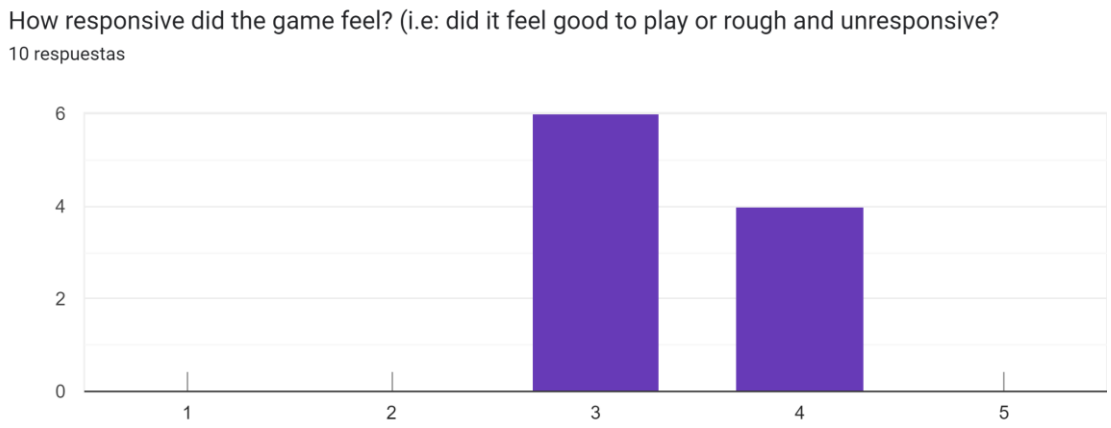
8 respuestas

Age of Empires because of farms being a critical building
yes
age of empires. Tematica
Age of Empires, BUILDINGS
Age of Empires
Warcraft and age of empires
Warcraft, Starcraft, Age of Empires(but faster)
Yes, the visuals have reminded me of Warcraft and the gameplay is a bit similar to Age of Empires

Como podemos observar a todos los jugadores les ha parecido el juego similar a *Age of Empires* y en menor medida al *Warcraft*, en el apartado 2.1.5 se explicó que *Castle Strike* está inspirado en estos dos juegos de lo cual se puede concluir que en este aspecto el proyecto ha sido un éxito.

Las siguientes preguntas se evalúan en una escala del 1 al 5, siendo 1 la respuesta más negativa y 5 la más positiva.

Cuarta Pregunta: ¿Cómo de responsivo es el juego? ¿Se siente bien al jugar?

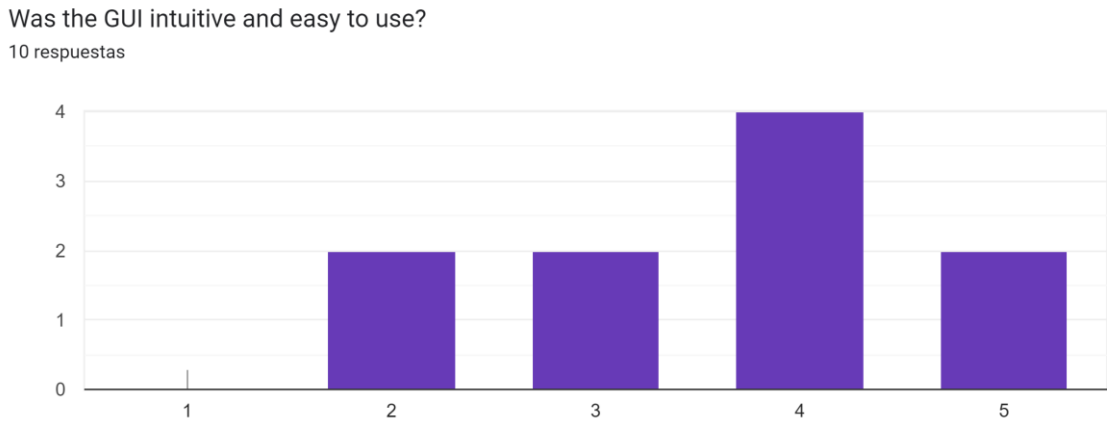


Aunque esta pregunta es muy subjetiva y depende la experiencia con otros RTS y de las expectativas de los jugadores, al estar el 60% de las respuestas en la opción 3 y el 40% en la opción 4 se puede concluir que la sensación de los jugadores es mayormente positiva, aunque se puede mejorar.

Quinta Pregunta: ¿Cómo es el feedback del juego en respuesta a tus inputs?

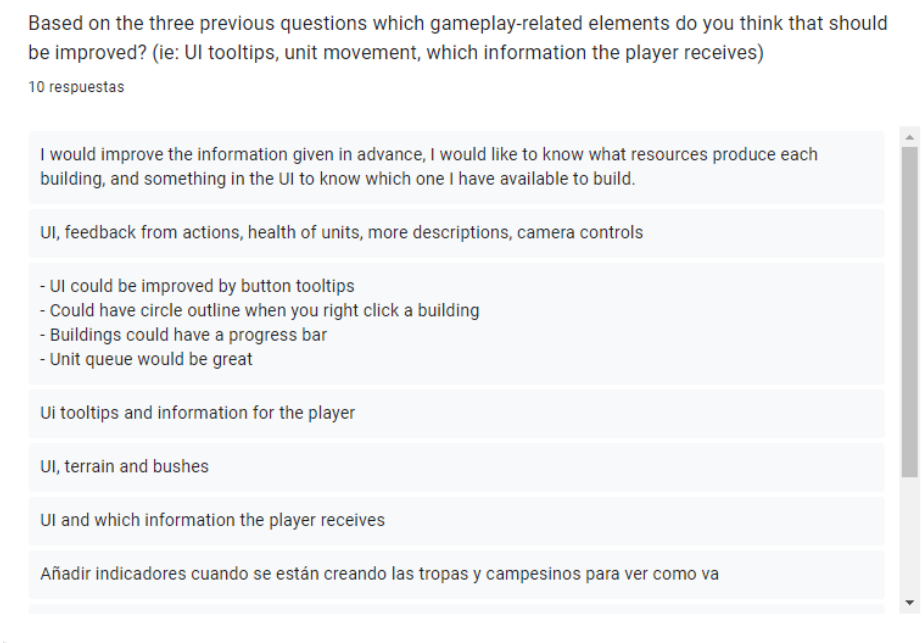
Esta pregunta casa con algunas de las valoraciones del apartado anterior respecto a que el juego no proporciona suficiente información al jugador, como podemos observar el 40% de las respuestas están en la opción 2.

Sexta Pregunta: ¿La interfaz es intuitiva y fácil de utilizar?



A pesar de las respuestas a la pregunta anterior, en esta podemos observar una mayoría positiva. Esto puede significar que la interfaz del juego es buena y cumple su función adecuadamente, pero es incompleta y no proporciona suficiente información.

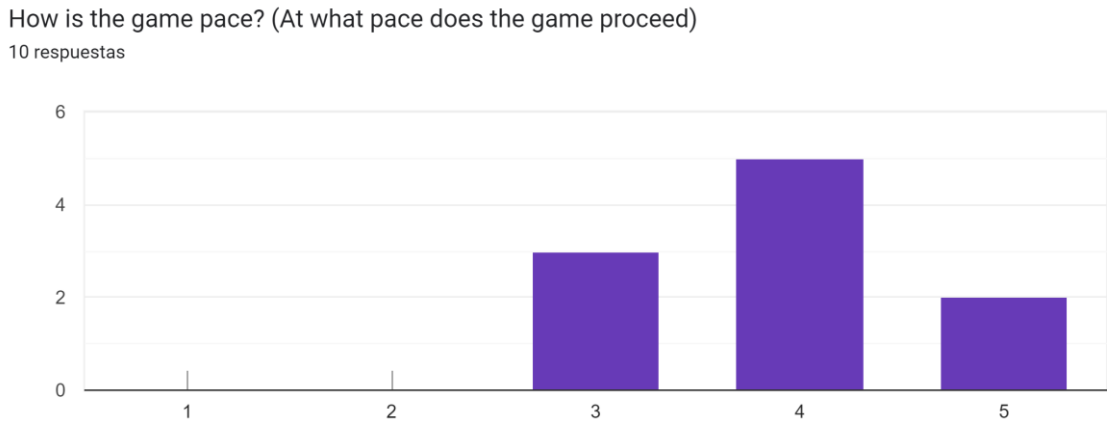
Séptima Pregunta: Basado en las tres preguntas anteriores ¿Qué elementos del juego crees que se deben mejorar?



Como era de esperar viendo las respuestas anteriores todos los jugadores concuerdan en que el juego no proporciona suficiente información y proponen distintos cambios. Los más repetidos son:

- Mostrar la cola de reclutamiento en los edificios para ver que se está reclutando.
- Añadir ventanas de información al pasar el ratón sobre los iconos de construcción o reclutamiento que indiquen el coste de recursos y que es lo que se desea crear.
- Alertas para el jugador.

Octava Pregunta: ¿Cómo de rápida es la partida?



En esta pregunta la escala representa lo rápida o lenta que transcurre la partida, donde 1 es demasiado lenta y 5 demasiado rápida. Para analizar la respuesta a esta pregunta se va a complementar con la duración registrada por Unity Analytics.

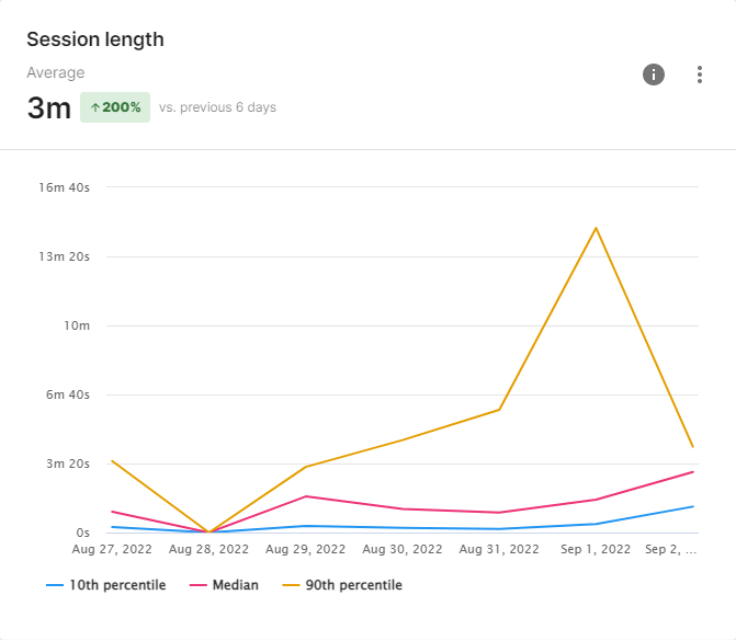
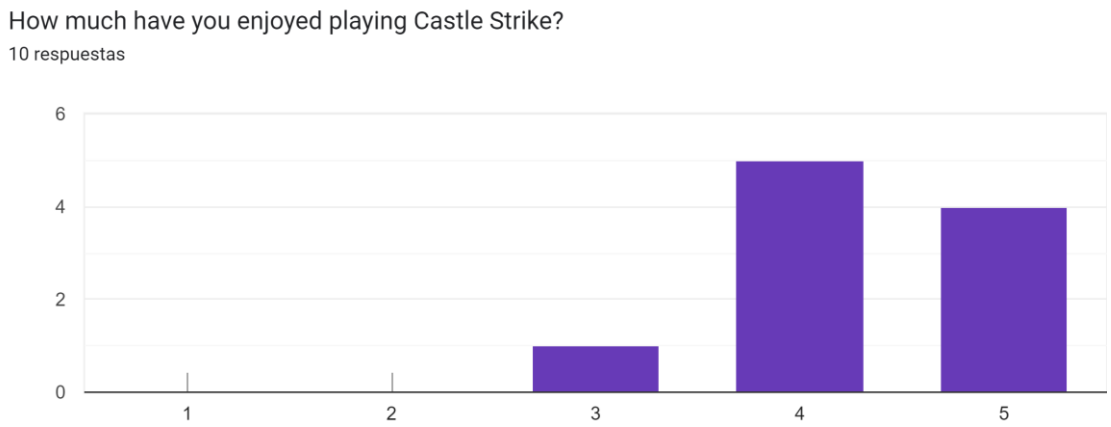


Figura 41 Duración de sesión de juego

Nótese que la *Figura* muestra 3 minutos como duración media de las sesiones de juego. Este dato no debe ser tenido en cuenta debido a que se incluyen todas las veces que se ha ejecutado el juego. Incluyendo ejecuciones muy cortas de unos pocos minutos mientras se probaban funcionalidades durante el desarrollo. Se debe prestar atención a la fecha del 1 de septiembre en la gráfica, que muestra un aumento considerable en la duración de las sesiones porque es cuando se realizaron las pruebas con los usuarios.

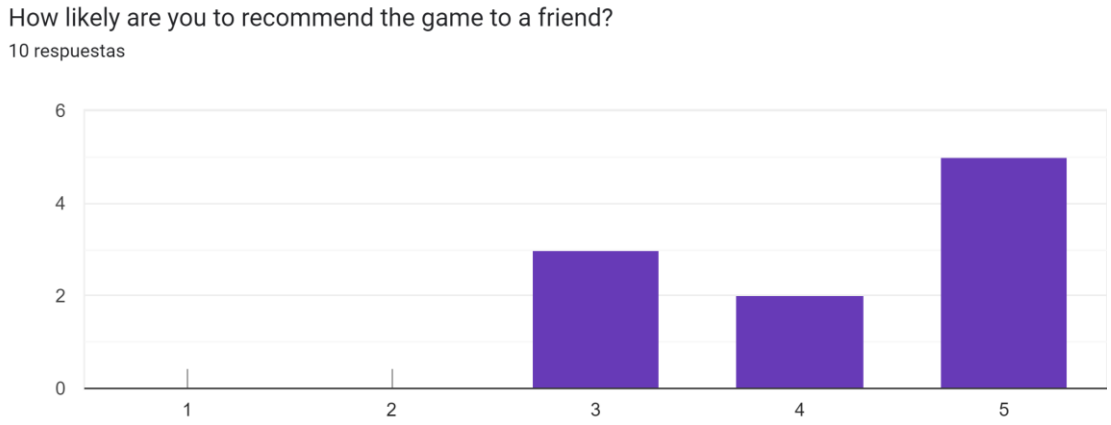
La duración media de las sesiones de juego durante las pruebas fue de 14m 42s y en las respuestas a la pregunta el 50% de jugadores han respondido 4 y el 20% 5, por lo que se puede extraer la conclusión de que las partidas transcurren demasiado rápido y que los jugadores esperaban que durasen más.

Novena Pregunta: ¿Cuánto te has divertido jugando a *Castle Strike*?



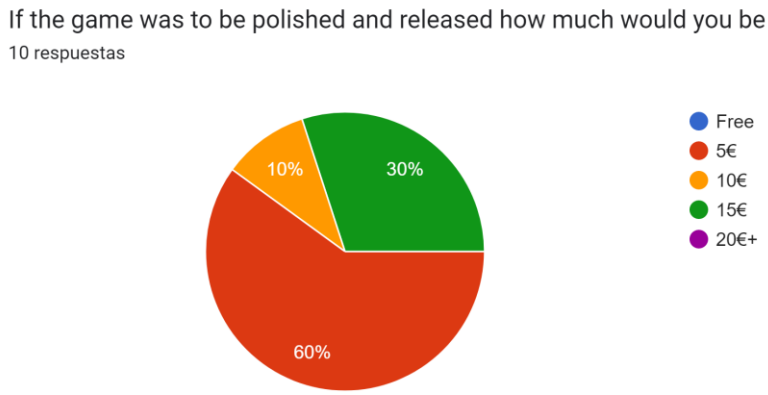
Como se concluyó en el apartado anterior el juego es divertido y las respuestas a esta pregunta lo corroboran.

Décima Pregunta: ¿Cómo de probable es que le recomiendes el juego a un amigo?



En esta pregunta el 70% de las respuestas son de 4 o superior mostrando que *Castle Strike* es un buen juego y que para ser un proyecto pequeño de un único desarrollador quizá podría alcanzar cierto grado de éxito en el mercado.

<u>Undécima Pregunta: Si el juego se terminara y fuera lanzado al mercado ¿Cuánto estarías dispuesto a pagar por él?</u>



El 60% de los jugadores pagarían 5€ por el juego, el 30% pagarían hasta 15€ y el 10% pagaría 10€, junto con las respuestas a la pregunta anterior confirma que el juego de ser lanzado podría obtener beneficios.

7. Conclusiones

Los resultados conseguidos con el desarrollo de este proyecto son sin duda alguna satisfactorios. Se han alcanzado todos los objetivos propuestos y el resultado ha sido un videojuego que con un poco más de trabajo podría publicarse al mercado.

Gracias a este trabajo se han reforzado conceptos de informática y del desarrollo ágil y se han obtenido conocimientos aplicables a nivel profesional. El proyecto ha sido desafiante y ha requerido amplio aprendizaje para poder completarlo, desde el uso de herramientas avanzadas de Unity a todo lo necesario para el juego online.

7.1 Relación del trabajo con los estudios cursados

Este trabajo ha requerido aplicar conceptos estudiados a lo largo de toda la carrera. Desde conceptos básicos como la herencia, uso de interfaces, diseño de interfaces de usuario y estructuras de datos, así como conocimientos estudiados en asignaturas optativas de desarrollo de videojuegos como optimización en Unity y conceptos de la rama de ingeniería de software como los code smells y técnicas de refactorización que se han aplicado durante el desarrollo. Además, al ser una aplicación distribuida, se ha tenido que lidiar con la sincronización de datos.

También se ha empleado la metodología ágil en el desarrollo del proyecto tal y como se ha estudiado en las asignaturas de Proceso de Software y Proyecto de ingeniería de software, y se han usado técnicas de validación con usuarios como el testeo de la aplicación y las encuestas tal y como se ha estudiado también en la rama.

El proyecto ha permitido aumentar mis conocimientos de Unity a un nivel muy superior al utilizado en las asignaturas optativas del grado y he adquirido conocimientos como el crear componentes personalizados en Unity, el funcionamiento de la herramienta Terrain para crear mapas rápidamente y con poco esfuerzo… También se ha aprendido a utilizar PUN y los conceptos fundamentales para el desarrollo de videojuegos online como el uso de RPCs, matchmaking para emparejar a los jugadores en partidas y autenticación de usuarios, aunque no se ha utilizado en el proyecto. Se han estudiado también las distintas arquitecturas que existen para videojuegos multijugador, sus ventajas y desventajas y para qué tipo de juegos es mejor una que otra.

7.2 Trabajos futuros

El videojuego en su estado actual podría considerarse que está en versión alfa, es decir, todas sus mecánicas son funcionales y la aplicación es estable, pero está incompleta. Existen mejoras que se pueden realizar en el videojuego que, o bien se han quedado en el backlog, o directamente se descartaron por falta de tiempo. Un videojuego sigue siendo un software muy grande y complejo que se suele desarrollar en equipos de decenas o, incluso, en las mayores producciones, cientos de personas. Requiere de muchas disciplinas tanto técnicas como artísticas, desde la programación, modelado 3D hasta conocimientos musicales. Por mi parte planeo continuar con el desarrollo del juego en mi tiempo libre y añadirle un modo de un solo jugador con múltiples niveles,

añadir más unidades y edificios y una vez me parezca un producto completo publicarlo para poder recibir valoraciones de un público más amplio.

Los videojuegos tienen como objetivo principal entretener, ser divertidos, es por ello que colocan al usuario en el centro de su funcionamiento. Desde requisitos funcionales como tiempos de respuesta de los controles a opciones de accesibilidad para personas con discapacidades visuales o motoras. Por ello, muchas prácticas comunes en el desarrollo de videojuegos se podrían aplicar a cualquier otro tipo de software para hacerlo mejor. Un ejemplo son interfaces de usuario para personas con daltonismo que, fuera del ámbito de los videojuegos, es muy raro encontrar software que incluya opciones de este tipo.

Para concluir, el proyecto ha aumentado mis conocimientos de programación y me ha servido para reforzar mi convicción de que escogí la carrera universitaria correcta. Además, ha despertado en mí el interés de aprender modelado y animación 3D para poder realizar futuros proyectos en Unity sin depender de si existe o no el asset que necesito en la Asset Store.

8. Referencias

[1] DESARROLLO ESPAÑOL DE VIDEOJUEGOS. *Libro blanco del desarrollo Español de videojuegos 2020* [en línea]. España: Madrid, Abril 2021 [fecha de consulta 4 marzo 2022]. Disponible en: https://www.dev.org.es/es/publicaciones/libro-blanco-dev-2020

[2] J. P. WOLF Mark. *Encyclopedia of Video Games: The Culture, Technology and Art of Gaming*. 1ª ed. Santa Barbara, California, 2012, pp. 629. ISBN 978-0-313-37936-9

[3] J. P. WOLF Mark. *Encyclopedia of Video Games: The Culture, Technology and Art of Gaming*. 1ª ed. Santa Barbara, California, 2012, pp. 630. ISBN 978-0-313-37936-9

[4] J. P. WOLF Mark. *Encyclopedia of Video Games: The Culture, Technology and Art of Gaming*. 1ª ed. Santa Barbara, California, 2012, pp. 695. ISBN 978-0-313-37936-9

[5] J. P. WOLF Mark. *Encyclopedia of Video Games: The Culture, Technology and Art of Gaming*. 1ª ed. Santa Barbara, California, 2012, pp. 624-626. ISBN 978-0-313-37936-9

[6] Colaboradores de Wikipedia. *Risk* [en línea]. *Wikipedia, La enciclopedia libre*, 2022 [fecha de consulta 9 de junio de 2022]. Disponible en: https://es.wikipedia.org/wiki/Risk

[7] *Pong Game* [en línea] [fecha de consulta 9 junio de 2022]. Disponible en: https://www.ponggame.org/

[8] *Civil War* [en línea] [fecha de consulta 15 junio de 2022]. Disponible en: https://www.mobygames.com/game/pet/civil-war_

[9] AHL David. *Hammurabi* [en línea] [fecha de consulta 16 junio de 2022]. Disponible en: https://www.atariarchives.org/basicgames/showpage.php?page=78

[10] *M.U.L.E.* [en línea] [fecha de consulta 15 junio de 2022]. Disponible en: https://www.mobygames.com/game/atari-8-bit/mule

[11] BOUZO Oscar. *Todos los juegos de la saga Age of Empires ordenados de peor a mejor* [en línea] [fecha de consulta 16 junio de 2022]. Disponible en: https://www.vidaextra.com/listas/todos-juegos-saga-age-of-empires-ordenados-peor-a-mejor

[12] ADAMS Dan. *Warhammer 40,000: Dawn of War Review* [en línea] [fecha de consulta 16 junio de 2022]. Disponible en: https://www.ign.com/articles/2004/09/21/warhammer-40000-dawn-of-war-review?page=3

[13] ROSSIGNOL Jim. *Star Wars: Empire At War* [en línea] [fecha de consulta 16 junio de 2022]. Disponible en: https://www.eurogamer.net/r-starwarseaw-pc

[14] KASAVIN Greg. *Company of Heroes Review* [en línea] [fecha de consulta 16 junio de 2022]. Disponible en: https://www.gamespot.com/reviews/company-of-heroes-review/1900-6157430/

[15] FUNK John. *MOBA, DOTA, ARTS: A brief introduction to gaming's biggest, most impenetrable genre* [en línea] [fecha de consulta 22 junio de 2022]. Disponible en: https://www.polygon.com/2013/9/2/4672920/moba-dota-arts-a-brief-introduction-to-gamings-biggest-most

[16] Connected Games: Building real-time multiplayer games with Unity and Google – Unite LA [en línea] Presentado por Micah Baker y Brandi House, 11 Noviembre 2018 [fecha de consulta 19 junio de 2022]. Disponible en: https://youtu.be/CuQF7hXlVyk

[17] *¿Qué significan las etiquetas?* [en línea] [fecha de consulta 23 junio 2022]. Disponible en: https://pegi.info/es/que-significan-las-etiquetas

[18] PALACIOS Angela. *Low Poly: el arte de crear personajes y escenas con polígonos* [en línea] [fecha de consulta 30 junio 2022]. Disponible en: https://www.crehana.com/blog/animacion-modelado/low-poly/

[19] *Asset Store Terms of Service and EULA* [en línea] [fecha de consulta 30 junio 2022]. Disponible en: https://unity3d.com/legal/as_terms

[20] *Navigation System in Unity* [en línea] [fecha de consulta 13 julio 2022]. Disponible en: https://docs.unity3d.com/Manual/nav-NavigationSystem.html

[21] *Inner Workings of the Navigation System* [en línea] [fecha de consulta 13 julio 2022]. Disponible en: https://docs.unity3d.com/Manual/nav-InnerWorkings.html

[22] *Algoritmo de Búsqueda Heurística A** [en línea] [fecha de consulta 2 septiembre 2022]. Disponible en: https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*

[23] STEWART Nick. *Metal Fatigue PC Review* [en línea] [fecha de consulta 16 junio 2022]. Disponible en: https://archive.ph/20120719182031/http://www.avault.com/reviews/pc/metal-fatigue/3/

[24] *Choosing the Right Netcode* [en línea] [fecha de consulta 16 junio 2022]. Disponible en: https://create.unity.com/form-netcode-report

[25] Colaboradores de Wikipedia. *Dune II* [en línea]. *Wikipedia, La enciclopedia libre*, 2022 [fecha de consulta 27 agosto 2022]. Disponible en: https://es.wikipedia.org/wiki/Dune_II

9. Anexos

9.1 Game Design Document

Análisis del Juego

Castle Strike es un videojuego de estrategia 3D del género RTS en el cual se tiene que reclutar un ejército para derrotar al resto de jugadores enemigos.

El videojuego consistirá en un modo multijugador de 2 a 4 jugadores que lucharán por equipos o individualmente con el objetivo de destruir la base enemiga.

Declaración de objetivos

Utiliza tu genio estratégico para sorprender y vencer a tus enemigos empleando tácticas y estrategias superiores a las suyas.

Género y extensión del juego

RTS, multijugador. El juego se compondrá de varios mapas donde jugar las partidas.

Plataformas

Microsoft Windows, macOS, Linux.

Público Objetivo

Jugadores a partir de 12 años de edad a los que les guste el género de la estrategia.

PEGI

El juego está orientado a una clasificación de PEGI 12 debido a que existen muestras de violencia de una naturaleza gráfica hacia los personajes, pero sin ser parecido a la vida real.

Jugabilidad

Resumen de la Jugabilidad

Se trata de un videojuego perteneciente al género de la estrategia en tiempo real y estará disponible únicamente para PC.

Castle Strike será un videojuego multijugador en el que la inteligencia y capacidad de decisión del jugador serán su principal herramienta.

Experiencia del Jugador

Empiezas en un menú minimalista desde el que se accede al menú multijugador donde se mostrará un listado de las partidas existentes o se podrá crear una nueva a la que podrán unirse otros jugadores, tras configurar una partida se empezará y cada jugador controlará su base y sus tropas.

Directrices de Jugabilidad

El juego se enfocará en la construcción de bases y el combate por destruir las del resto de jugadores por lo que se deberá diseñar el juego teniendo en cuenta el equilibrio entre las distintas tropas para que sea la habilidad de los jugadores la que les de la victoria.

El juego deberá incentivar la creatividad y la experimentación con distintas estrategias para permitir que no haya dos partidas iguales ni la existencia de una única estrategia ganadora.

Objetivos del Juego y Recompensas

Recompensas	*Obstáculos*	*Dificultad de los niveles*
Ganar partidas y subir de nivel de habilidad para enfrentarse a mejores jugadores.	El resto de los jugadores	La dificultad dependerá del propio nivel del jugador y de la habilidad de su contrincante ya que el que sea el mejor estratega vencerá.

Mecánicas del Juego

Atributos de las unidades	
Campesino	- Construir: los campesinos son la única unidad capaz de construir nuevos edificios para la base. - Recolección de recursos: oro, madera, piedra y comida son los recursos que los campesinos recolectarán. - Milicia: en caso de necesidad los campesinos pueden transformarse temporalmente en milicianos, más débiles que soldados normales, para luchar contra las tropas enemigas.
Espadachín	- Muro de escudo: el espadachín avanza con su escudo alzado, reduciendo su velocidad de movimiento y aumentando su defensa. -
Arquero	- Salva: los arqueros concentran sus disparos en una zona para crear una zona por donde resulta imposible avanzar sin sufrir graves bajas.
Caballero	- Carga: los caballeros aceleran a gran velocidad con sus monturas y embisten a los soldados enemigos infligiendo grandes daños.
Catapulta	- Asedio: las catapultas causan daño aumentado a las estructuras.

Batalla	Modo de juego único multijugador.
Conquista	Ganará el jugador que destruya el edificio principal del enemigo.
Victoria total	Ganará el jugador que destruya todos los edificios de la base enemiga.

Diseño de Niveles

El videojuego se compondrá de un conjunto de mapas de estilo RTS con localizaciones fijas y distintos puntos de recursos por los que los jugadores deberán competir. Los mapas tendrán distintos tamaños en función del número de jugadores de la partida.

Cámara

La cámara en Castle Strike será una cámara clásica de RTS de estilo isométrico con zoom para acercarla y alejarla.

Inteligencia Artificial

Las unidades contarán con una IA para pathfinding así como con una IA básica para poder realizar acciones automáticas como la recolección de recursos, cuando se agote un nodo buscar otro del recurso al que estaba asignado el campesino, o atacar automáticamente a enemigos cercanos u ordenar defender una posición.

Esquema de controles

Acción	Teclado
Selección de unidades	Click izquierdo del ratón sobre la unidad
Selección múltiple de unidades	Mantener shift + click sobre las unidades o Mantener click izquierdo y arrastrar el ratón sobre las unidades
Movimiento	Click derecho sobre un punto válido del mapa
Movimiento múltiple	Mantener shift + Click derecho sobre los puntos a los que se quiera mover las unidades

Establecer un punto de reunión	Seleccionar un edificio de producción de unidades y hacer click derecho en un punto válido del mapa
Crear grupo de control	Con las unidades deseadas seleccionadas pulsar control + número
Menú	Escape

Estética del Juego e Interfaz de Usuario

Estética de Niveles

El juego será luminoso y tendrá una estética cartoon que encaje con el diseño de las unidades.

Estética de Personajes

Tanto las unidades como estructuras serán las contenidas en el Toony Tiny RTS set obtenido de la Unity Asset Store.

Representación general del paquete de Assets.

Diseño de audio

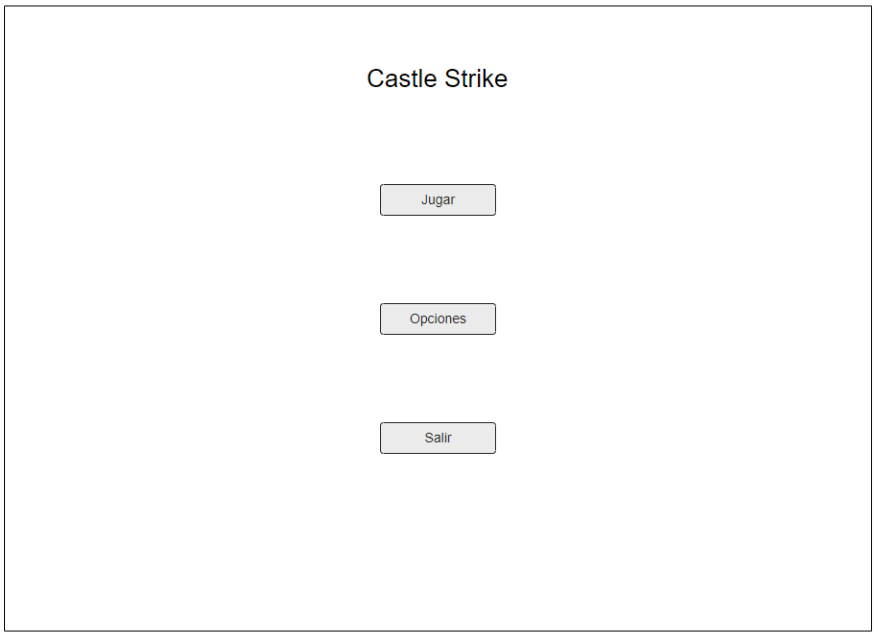
El juego cuenta con música diferente para los menús y para las partidas que ayuda a la inmersión y a caracterizar el juego, también hay efectos de sonido para las interacciones de los menús como en la transición entre estos como las interacciones con los elementos del menú de opciones, a su vez también hay efectos de sonido para las acciones de las unidades como talar

árboles o construir. El volumen tanto de la música como de los efectos puede ser regulado, independientemente el uno del otro, con los sliders del menú de opciones.

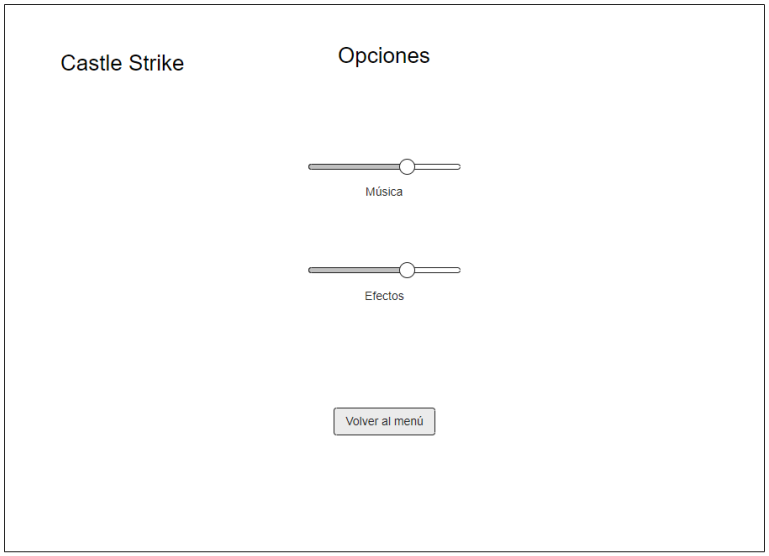
Interfaz de Usuario

Como solo hay un modo de juego, el menú principal permitirá al jugador ir al menú de jugar, cambiar las opciones (ajustes de sonido) o salir del juego.

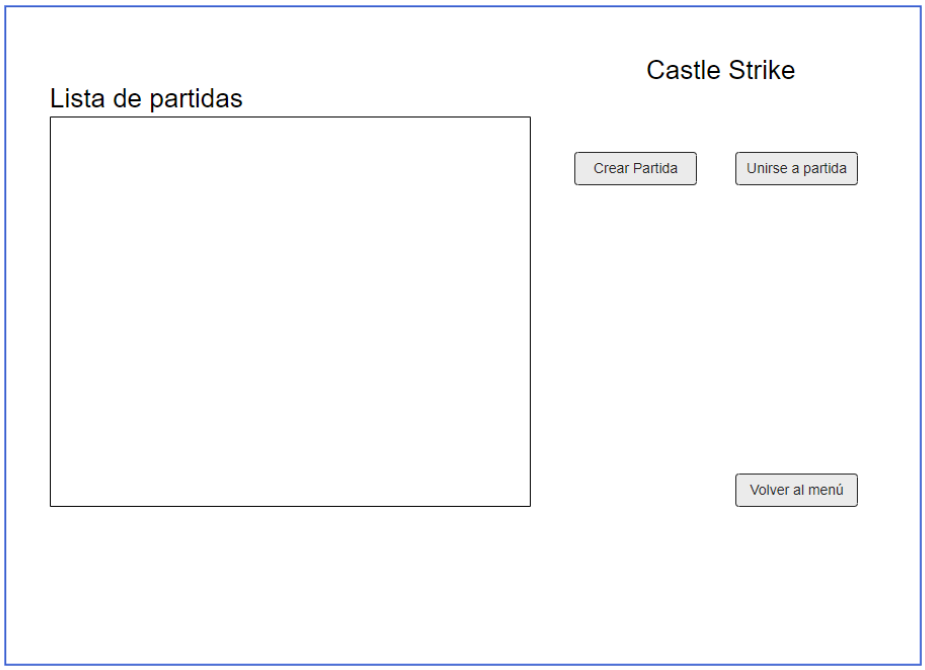
Respecto a la interfaz de la partida estará inspirada en la de juegos RTS clásicos como Warcraft 3 o Age of Empires, siendo similar a la de estos.



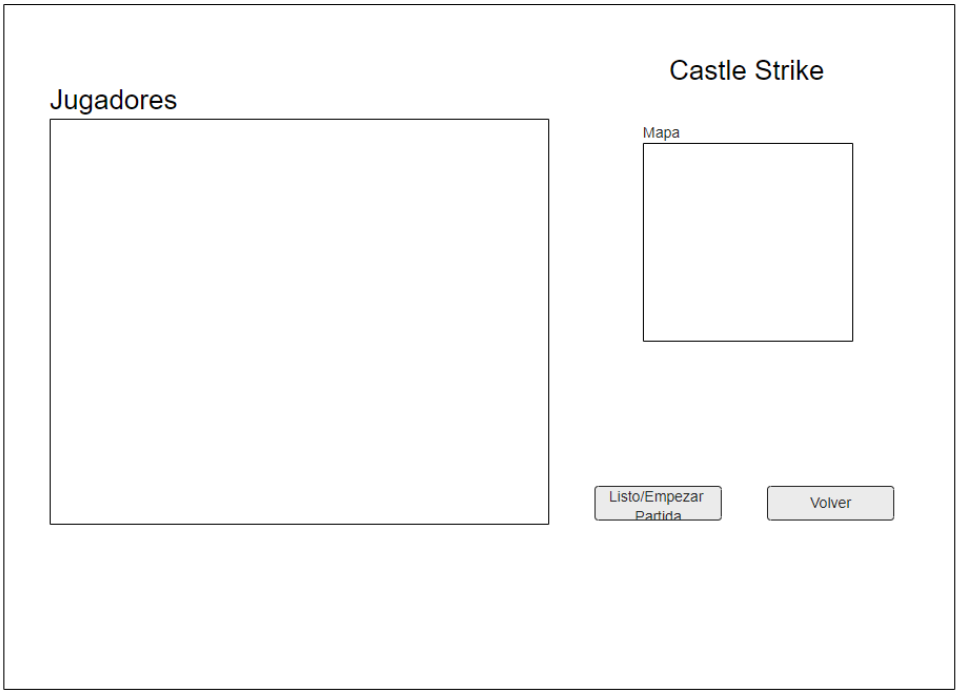
Mockup del menú principal



Mockup del menú de opciones



Mockup de la lista de selección de partida



Mockup del lobby de una partida

Interfaz del Warcraft 3.

Interfaz del Age of Empires 2.

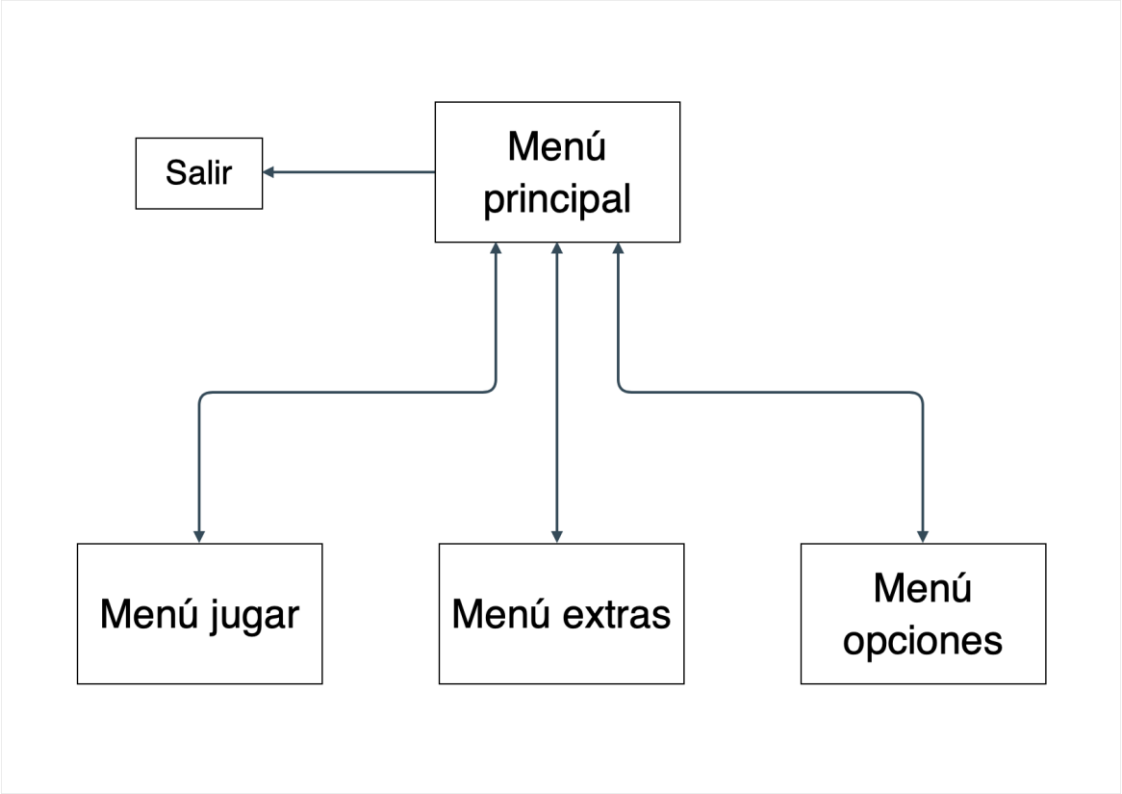


Diagrama de navegación entre menús.

9.2 Manual de uso de PUN

9.2.1 Configuración de PUN dentro del proyecto

El primer paso para poder utilizar PUN es crear una cuenta de Photon y registrar una aplicación en ella, este proceso es muy sencillo y no necesita de más de unos pocos minutos.

Tras esto el siguiente paso es añadir el asset de PUN al proyecto de Unity, proceso también sencillo. El asset se importa como cualquier otro desde el package manager de Unity. Después importarlo se abrirá la ventana de configuración de PUN, en esta tenemos que introducir el appId de la aplicación que hemos registrado en Photon en el paso anterior.

Una vez completados estos dos pasos ya se puede empezar el desarrollo de las características de PUN.

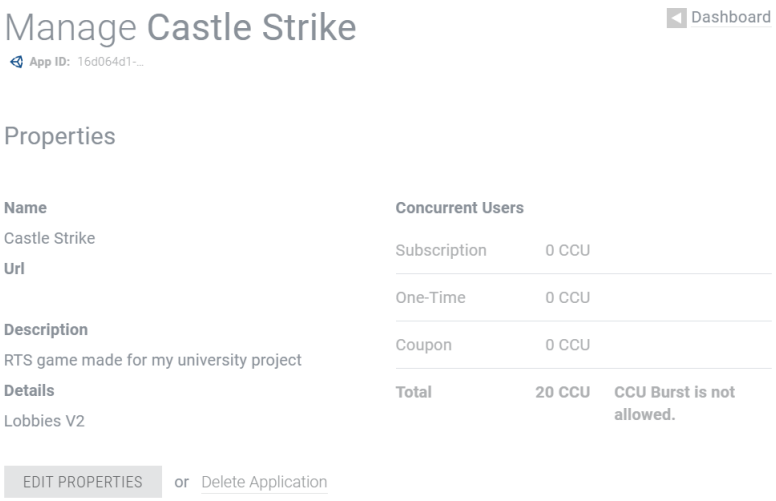


Figura 42 Vista de la aplicación en la web de Photon

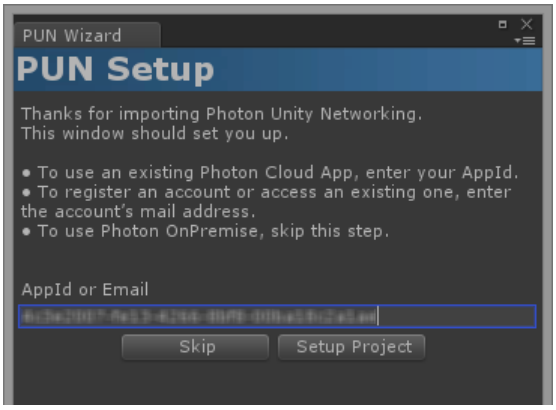


Figura 43 Ventana de configuración de PUN dentro de Unity

9.2.2 Clases principales de PUN

En este apartado se enumerarán las clases de PUN y sus funciones utilizadas en este proyecto.

- PhotonNetwork
  Es la clase principal de PUN, necesaria para conectarse y desconectarse a los servidores.
  Funciones principales:
  - *ConnectUsingSettings* para conectarse a los servidores, si se utiliza sin argumentos utiliza la configuración contenida en el archivo PhotonServerSettings.
  - *Disconnect* se desconecta de PUN, abandonando cualquier sala en el proceso.
  - *CreateRoom* crea una sala para alojar una partida del juego, toma como argumento obligatorio el nombre de la sala y opcionalmente la configuración de esta, como puede ser el número máximo de jugadores permitidos.
  - *JoinRoom* para unirse a una sala, toma como argumento el nombre de la sala.
  - *LeaveRoom* abandona una sala y regresa al lobby para crear o unirse a otras partidas.
  - *Instantiate* para instanciar objetos a través de la red, el jugador que llama al método se convierte en el dueño del objeto instanciado.

  - *Destroy* para destruir objetos a través de la red, tiene que llamarlo el dueño del objeto a destruir.
  - *LoadLevel* se utiliza para cargar el nivel de forma sincronizada entre todos los jugadores de la partida.
- PhotonView
  Componente de los gameObjects que los identifica en la red y configura como se sincroniza.
  - *OnPhotonSerializeView* se utiliza para sincronizar variables a través de la red. Funciona de forma unidireccional por lo que solo el dueño de la PhotonView puede enviar datos. El resto de los jugadores los reciben y aplican al objeto en cuestión localmente.
  - *Find* busca el objeto cuya id coincida con la especificada como argumento.
  - *RPC* realiza una llamada RPC a una función marcada como tal, recibe como argumento el nombre de la función a ejecutar como RPC.
  - *IsMine* propiedad que devuelve true para el dueño de la photonView. Necesaria para verificar que cada jugador solo tiene control sobre sus objetos.

- MonoBehaviourPunCallbacks
  Clase que implementa los eventos de PUN, hereda de la clase MonoBehaviour de Unity para poder utilizarse en los gameObjects y se utiliza en conjunción con la clase PhotonNetwork
  - *OnCreateRoomFailed* se llama cuando la creación de una sala falla, recibe de argumento el error para poder informar al jugador.
  - *OnConnectedToMaster* se llama tras conectarse a los servidores de Photon, útil para evitar que el jugador interactúe en la pantalla de búsqueda de partidas hasta que se conecta.
  - *OnJoinRoomFailed* se llama cuando se falla al unirse a una sala, recibe el error para poder informar al jugador.
  - *OnJoinedRoom* se llama tras unirse a una sala. En este proyecto se utiliza para actualizar la lista de jugadores en la sala.
  - *OnPlayerLeftRoom* se llama cuando un jugador abandona una sala en los clientes que permanecen en esta. En este proyecto se utiliza para actualizar la lista de jugadores en la sala.
  - *OnRoomListUpdate* se llama cuando se actualizan las salas existentes.

- RoomInfo
  Contiene la configuración de una sala.
  - *MaxPlayers* número máximo de jugadores para esta sala.
  - *IsOpen* define si pueden entrar nuevos jugadores a la sala.
  - *IsVisible* define si la sala es visible para los jugadores que no están en ella.

9.3 Objetivos de desarrollo sostenible

9.3.1 Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS)

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**				**X**
ODS 5. **Igualdad de género.**			**X**	
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**		**X**		
ODS 9. **Industria, innovación e infraestructuras.**				**X**
ODS 10. **Reducción de las desigualdades**.				**X**
ODS 11. **Ciudades y comunidades sostenibles**.				**X**
ODS 12. **Producción y consumo responsables.**				**X**
ODS 13. **Acción por el clima.**				**X**
ODS 14. **Vida submarina.**				**X**
ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

9.3.2 Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

La naturaleza de este trabajo no contempla la sostenibilidad como un objetivo a perseguir puesto que es un videojuego y, en todo caso, la infraestructura necesaria para la conservación y distribución del proyecto produce un impacto negativo en el medio ambiente. Además, este proyecto se inició antes de la inclusión como requisito en los Trabajos de Fin de Grado de los objetivos de desarrollo disponible.

Sin embargo, el trabajo podría relacionarse con el ODS 8 y en menor medida con el ODS 5.

Como se ha explicado al principio de este trabajo el sector de los videojuegos está en constante crecimiento en todo el mundo y proporciona trabajo a cada vez más personas[1]. Durante la crisis del COVID ha demostrado su robustez, al menos en España, ya que la mayoría de

estudios de videojuegos mantuvieron su plantilla e incluso el número de personas empleadas en este sector se incrementó. En un contexto de crisis económica internacional el pronóstico en el sector de los videojuegos es de crecimiento económico y se prevé que el número de personas empleadas en el sector continúe creciendo. Por estos motivos podemos concluir que este trabajo se relaciona con el ODS 8.

Por otra parte, como es común en muchas ramas de la informática, es un sector en el que la mayoría de puestos están ocupados por hombres, sin embargo, cada vez más mujeres se adentran en el mundo del desarrollo de videojuegos. Alrededor del 18,5% de los empleos en el sector de los videojuegos en España están ocupados por mujeres y la mayoría de estudios afirman tener en marcha planes de igualdad para aumentar esta cifra[1]. Desde el punto de vista del consumidor en el sector de los videojuegos hay una mayor igualdad, el 42% de *gamers* son mujeres lo cual es una noticia alentadora puesto que antiguamente los videojuegos se consideraban un entretenimiento principalmente masculino.

A pesar de haber desarrollado el trabajo sin considerar ODS dentro del sector de los videojuegos existen analistas y muchos estudios que los consideran a la hora de orientar sus procesos. Es por esto que el trabajo ha podido relacionarse con estos dos objetivos.

Glosario

Componente: Unidad que dota de una funcionalidad a un GameObject.

Escena: Escenario donde transcurre la partida.

E-sports: Competiciones profesionales de videojuegos multijugador.

FPS: Género de videojuegos que simula el uso de armas de fuego desde una perspectiva de primera persona

GameObject: Unidad básica para construir las escenas del juego.

GameObject Vacío: GameObject que solo contiene el componente transform.

Low-Poly: Técnica de modelado 3D que utiliza un número muy bajo de polígonos.

MMORPG: Juego de rol multijugador masivo online. Videojuego multijugador que se caracteriza por tener miles de jugadores en un mismo servidor.

MOBA: Multiplayer Online Battle Arena.

NavMesh: Estructura de datos que describe las superficies caminables del mundo del juego y permite encontrar un camino entre una posición y otra.

NavMesh Agent: Componente de los gameObjects que utiliza la NavMesh para desplazarse e incorpora la capacidad para evitar a otros agentes y obstáculos estáticos o en movimiento.

NavMesh Obstacle: Componente de los gameObjects que describe objetos que los agentes deben tratar como obstáculos y evitar al desplazarse.

Photon View: Componente básico de PUN que otorga una ID única al gameObject al que pertenece y permite que se sincronice a través de la red.

Prefab: GameObject prefabricado.

Ray-Tracing: Algoritmo de síntesis de imágenes que simula el comportamiento físico de la luz para alcanzar un mayor realismo.

RPC: En computación distribuida, es una técnica utilizada para ejecutar código en otra máquina distinta a la que realiza la llamada.

Tag: Referencia asignable a uno o más gameObjects.