Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de la arquitectura online de un videojuego cooperativo en Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Héctor Palau Francisco

Tutor: Javier Lluch Crespo

2020-2021

Resumen

Este proyecto grupal recoge el proceso de diseño y desarrollo de un prototipo de un videojuego cooperativo en 3D, siendo este del género de plataformas y disparos en tercera persona, y pasando por las fases de ideación, diseño de la jugabilidad, diseño de personajes y escenarios, modelado 3D y programación de uno de sus niveles. El videojuego gira en torno a dos robots, cada uno controlado por un jugador, que deberán completar misiones, poniendo especial énfasis en la cooperación entre ambos.

Esta memoria ocupa la parte de desarrollo de los servidores y su arquitectura, junto con parte de la funcionalidad del juego como el comportamiento de los enemigos.
También, se implementará una base para poder llevar a cabo el multijugador desarrollando mecanismos para poder intercambiar información entre jugadores, además de sincronizar los eventos, estados, acciones y posiciones de todos los jugadores y enemigos.

Palabras clave: videojuego. Unity, multijugador, cooperativo, online, tercera persona

Abstract

This group project, carried out by three people, includes the process of designing and developing a prototype of a cooperative video game in 3D, this being the genre of platforms and third-person shooting with light puzzle elements, and going through the ideation phases, gameplay design, character and scenery design, 3D modeling and programming in Unity of one of its levels. The video game revolves around two robots, each controlled by a player, who must complete missions, placing special emphasis on cooperation between the two.

This memory occupies the development part of the servers and their architecture, along with part of the functionality of the game such as the behavior of the enemies.
Also, a base will be implemented to be able to carry out the multiplayer by developing mechanisms to be able to exchange information between players, in addition to synchronizing the events, states, actions and positions of all the players and enemies.

Keywords: videogame, Unity, multiplayer, online, cooperative, third person

Índice de contenidos

Índice de imágenes

1. Introducción

El sector de los videojuegos está en auge, cada año son jugados por más gente y generan más beneficios económicos. A su vez también crece el número de competiciones y el interés por deportes electrónicos, estos son competiciones de videojuegos multijugador.

Además, si se observan los 10 videojuegos más populares del año 2021, se podrá ver que son todos multijugador lo que indica que, en su mayoría, la población prefiere este tipo de títulos en los que hay más jugadores presentes en la partida y pueden competir. [1]

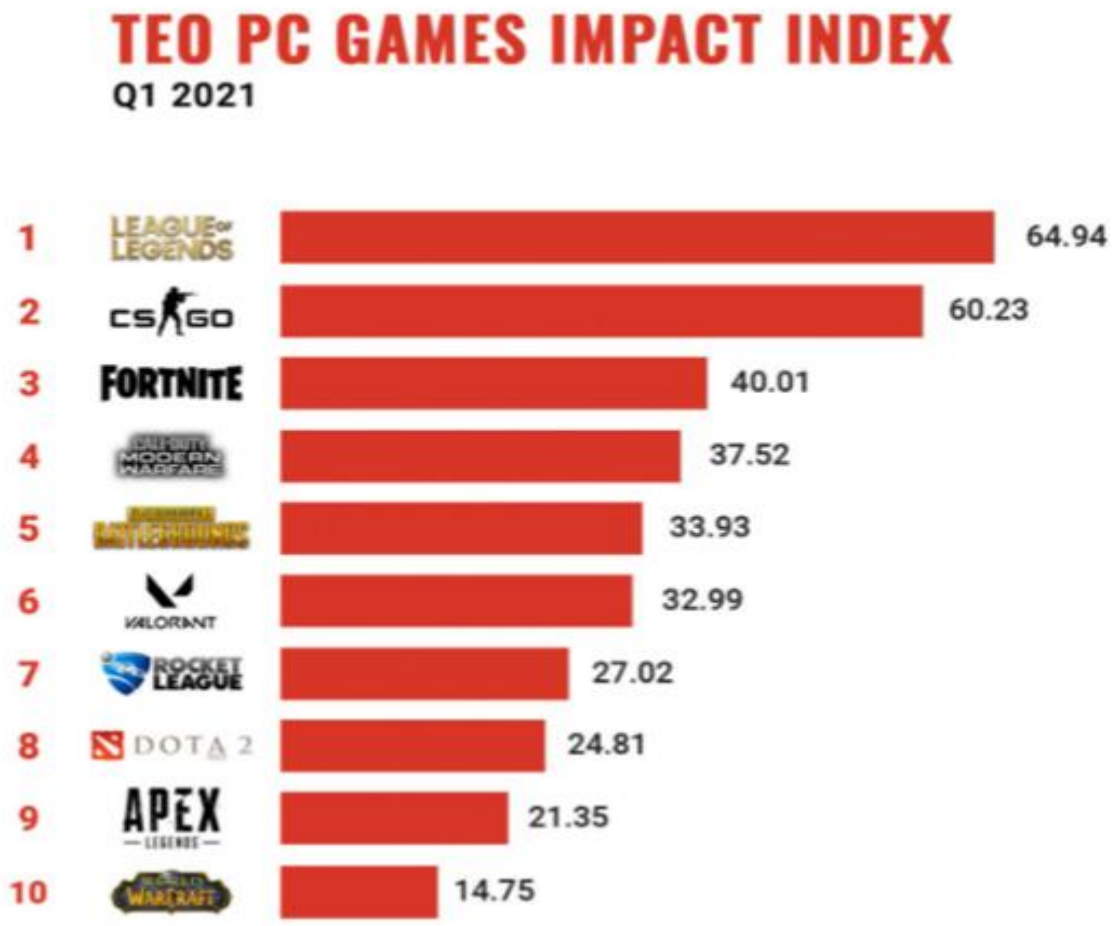

Ilustración 1 Los 10 juegos de ordenador con más ingresos en 2021.

Debido a que en estos juegos el usuario no está solo, los mismos suelen incorporan sistemas anti trampas que permitan que el juego sea justo para todos y evitan que algún jugador estropee la experiencia a los demás, dado que muchos de los títulos son competitivos no se puede permitir alguna ventaja injusta. Jugar con más gente también

implica que las acciones realizadas no sean instantáneas, haciendo que se produzca un retraso desde que realizamos una acción hasta que los demás jugadores también, por lo que se quiere una experiencia de juego fluida se deberá integrar algún sistema que palie este problema.

Por ello se ha decidido llevar a cabo el desarrollo de un videojuego online cooperativo en el que dos jugadores tendrán que cooperar para poder completar el juego correctamente.

Este proyecto ha sido realizado por tres personas diferentes, una de ellas realizó todo el diseño, otra realizó la integración de toda la parte visual en Unity junto a parte de las mecánicas del juego, y finalmente en esta memoria se llevará a cabo toda la parte multijugador junto al desarrollo de los servidores y que arquitectura elegir para permitir que estos sean escalables fácilmente, movimiento y sincronización de enemigos y diversos sistemas para mejorar la experiencia de juego como un sistema anti trampas o predicción de movimiento para que las acciones realizadas parezcan instantáneas.

El videojuego ha sido desarrollado en Unity y el código hecho en c#, siendo la plataforma objetiva Windows debido a que es el sistema operativo más usado para jugar, aunque siendo fácilmente exportable a diversas plataformas como Android o Linux.

1.1 Motivación

La principal motivación de este proyecto es llevar a cabo el desarrollo de un videojuego multijugador con una arquitectura cliente-servidor fácilmente escalable, alejándose así del P2P, lo que permite un juego más justo en el que se evitan todo tipo de trampas, siendo así apto para que diversos jugadores compitan entre sí.

Debido a que la API empleada por Unity para desarrollar videojuegos multijugador conocida como HLAPI fue marcada como obsoleta sin dar ninguna alternativa, en este proyecto también se pretende ofrecer una solución alternativa que permita conectar fácilmente a diferentes jugadores de una manera sencilla y a su vez sea lo suficientemente potente como para poder gestionar varias partidas simultaneas.

Al tener un servidor también evitaremos que la experiencia de juego se vea afectada por la mala conexión de otro jugador, problemas de conexión como no poder jugar con alguien por no tener los puertos del rúter abiertos y que no pueda conectarse con el jugador que haya creado la partida.

1.2 Estructura

La memoria quedara dividida en nueve apartados principales.

En el primer apartado **Introducción**, como ya se ha visto, se hará una breve introducción al mundo de los videojuegos y se explicará la estructura de la memoria.

En el siguiente, **Objetivos**, se definirán las metas a cumplir durante el desarrollo del proyecto que deben completarse para darlo por finalizado.

Mas adelante, en **Estado del arte**, se comparará el trabajo a desarrollar con las opciones existentes, ya sean videojuegos o sistemas para la transmisión de paquetes en un juego multijugador.

Una vez definidas las alternativas presentes, en **Planificación** se detallará un plan para conseguir el correcto desarrollo y finalización del proyecto. Siendo esta parte realmente importante puesto que será llevado a cabo por tres personas diferentes.

Mas adelante, en **Análisis y diseño**, una vez definidos los objetivos se estudiará la mejor manera para llevarlos a cabo, además de introducir los recursos a usar para conseguir la finalización exitosa del proyecto.

Luego, en **Implementación,** se detallará el desarrollo de todo lo implementado, desde la arquitectura hasta el propio videojuego, así como los problemas que surgen y las soluciones aplicadas.

Cuando ya se haya explicado toda la implementación, en el apartado **Pruebas,** se explicarán los diferentes test que se hicieron para evaluar si el sistema se comportaba de la manera esperada.

Cerca del final, en **Conclusión,** se hará una evaluación general del proyecto y se hablará de si se ha conseguido realizar todo lo planificado.

Finalmente, en Futuro, se introducirán algunas nuevas funcionalidades o mejoras que se podrían introducir en el proyecto

A parte, se incluye una **Bibliografía** con toda la información consultada, un **Glosario** para entender mejor conceptos relacionados con todo el desarrollo llevado a cabo y un **Anexo** con recursos útiles y el código más relevante.

2. Objetivos

En este proyecto se pueden distinguir dos objetivos claros, por una parte, generar una experiencia positiva en el jugador y lograr crear un sistema por el cual tengan que cooperar para conseguir avanzado. Por otra parte, se busca conseguir que los usuarios pueden jugar entre si sin problemas, además de impedir la realización de trampas por un cliente malicioso y llevar a cabo la implementación de diversos mecanismos que permitan paliar los problemas típicos de la red como perdida o retraso de paquetes.

2.1 Objetivos del videojuego

Primero, desarrollar un videojuego divertido, no se pretende desarrollar un producto completo sino un prototipo con funciones básicas que permitan mostrar las posibilidades de todo lo que se puede hacer.

El videojuego consistirá en la cooperación de dos jugadores para conseguir superar el nivel. Habrá una serie de enemigos que intentarán matarlos mientras que los jugadores podrán disparar sus armas para acabar con ellos. Para que no sea monótono se ha decidido añadir diversas armas y enemigos generando así una partida diferente cada vez.

En el videojuego se busca la cooperación por lo que los jugadores también tendrán que cooperar en diversas tareas como pulsar dos botones simultáneamente, haciendo así que busquen una manera de sincronizar sus acciones. También habrá otras formas de cooperación como que un jugador debe mantener un botón apretado que permite al otro continuar. Finalmente habrá un enemigo final al el que tendrán que dispararle en dos puntos concretos a la vez si desean acabar con él.

Se busca un juego divertido por lo que no será excesivamente difícil y al morir el jugador regresará a una serie de puntos predefinidos que se van desbloqueando conforme se avanza en el nivel, además hay una serie de armas escondidas por el mapa beneficiando a los usuarios que exploren el escenario más a fondo.

Lo último que se busca es que funcione en la mayoría de ordenadores, por lo que tiene que tener un diseño gráfico simple, consiguiendo así una fluidez que no entorpezca la experiencia del usuario. Haciendo hincapié en la cámara se mueva de una forma natural y permita al usuario jugador un largo tiempo sin experimentar una sensación de malestar.

2.2 Objetivos de la parte multijugador

Explorar las diferentes opciones de hacer un juego multijugador, proponiendo soluciones para los retos más habituales que se pueden encontrar al llevar a cabo un juego de estas características.

Por lo tanto, se llevará a cabo el desarrollo de una arquitectura compatible con un gran número de jugadores, permitiendo ampliarla fácilmente añadiendo más servidores si es necesario. También será necesario que esta permita identificar a los usuarios maliciosos para poder tener una competición justa.

Debido a esto se tendrá que encontrar que encontrar una manera de transmitir paquetes por la red y conseguir que todos los jugadores tengan sus estados sincronizados, es decir, que cada acción que se realice sea transmitida y sincronizada con los demás sin generar estados incoherentes.

Finalmente, también implementaremos un sistema de predicción de movimiento que permitirá ver moverse a los demás jugadores de manera fluida, aunque tengan problemas de conexión o su latencia en la red sea muy elevada.

3. Estado del arte

Como ya se indicó en la introducción, el sector de los videojuegos es la industria dominante en el sector del entretenimiento y está en auge constante. No solo en facturación sino en recursos humanos, con cada día más gente trabajando en el sector, como programadores, diseñadores o guionistas.

Debido al desarrollo de motores muy potentes como Unity, hoy en día un grupo reducido de personas puede crear un juego que hace años necesitaría muchos más profesionales, por lo que cada vez hay más gente interesada en llevar a cabo su obra.

A continuación, podemos ver la facturación de este sector y la comparación con diversos sectores de entretenimiento. [2]

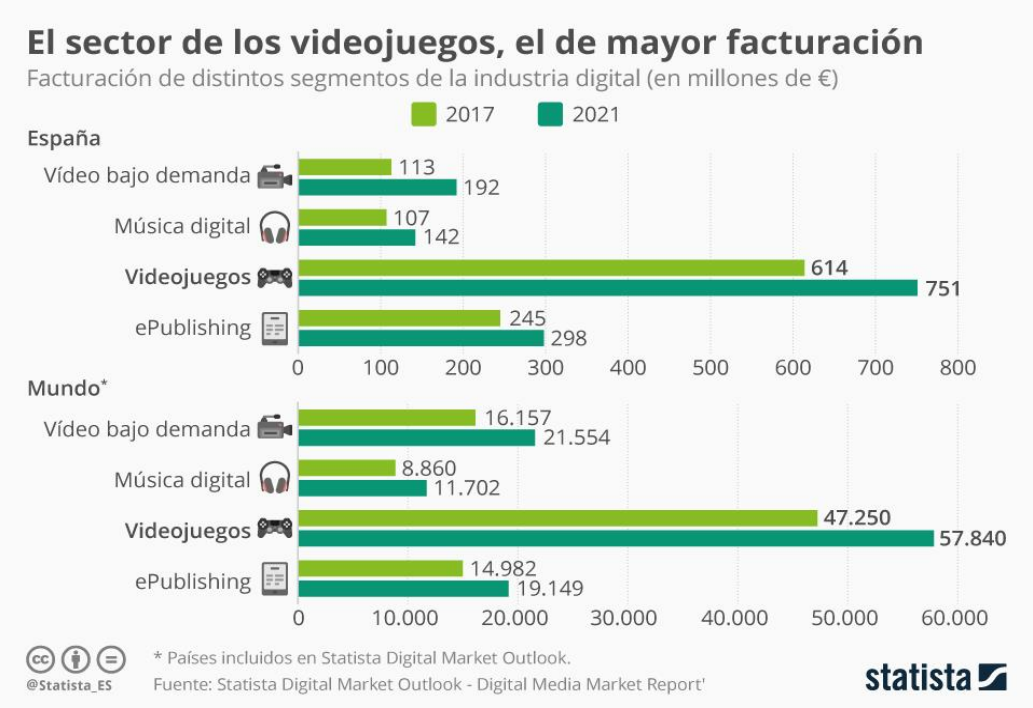

Ilustración 2 Ingresos de las diferentes industrias del entretenimiento.

Ilustración 3 Ingresos en la industria del videojuego.

Primero se compararán algunos juegos parecidos al proyecto que queremos desarrollar.

3.1 Videojuegos cooperativos multijugadores parecidos

Sobre el desarrollo llevado a cabo se pueden distinguir dos características principales, es un juego cooperativo y online por lo que se compararán juegos con estas características que han resultado de interés para la realización del proyecto.

3.1.1 **Portal 2**

Portal 2 es un videojuego que mezcla disparo en tercera persona con la resolución de puzles. Fue desarrollado por Valve Corporation.

Este videojuego se basa en ir resolviendo puzles mediante la utilización de una pistola que permite crear portales por los que podremos entrar y aparecer en el otro lado del portal.

Es considero uno de los mejores juegos del 2011 (año en el que se lanzó) y tiene elevadas puntuaciones en la mayoría de medios críticos.

Este juego incluía un modo cooperativo en el que cada jugador manejaba a un robot y mediante el uso de elementos del juego como geles, palancas o túneles y el trabajo en equipo podían superar los diferentes niveles. [3]

Ilustración 4 Videojuego portal 2.

3.1.2 **League of Legends**

El League of Legends es un videojuego del género multijugador de arena de batalla en línea el cual fue desarrollado por Riot Games.

Consiste en dos equipos de cinco jugadores cada uno que tienen que enfrentarse entre sí para conseguir destruir la base del equipo rival. [4]

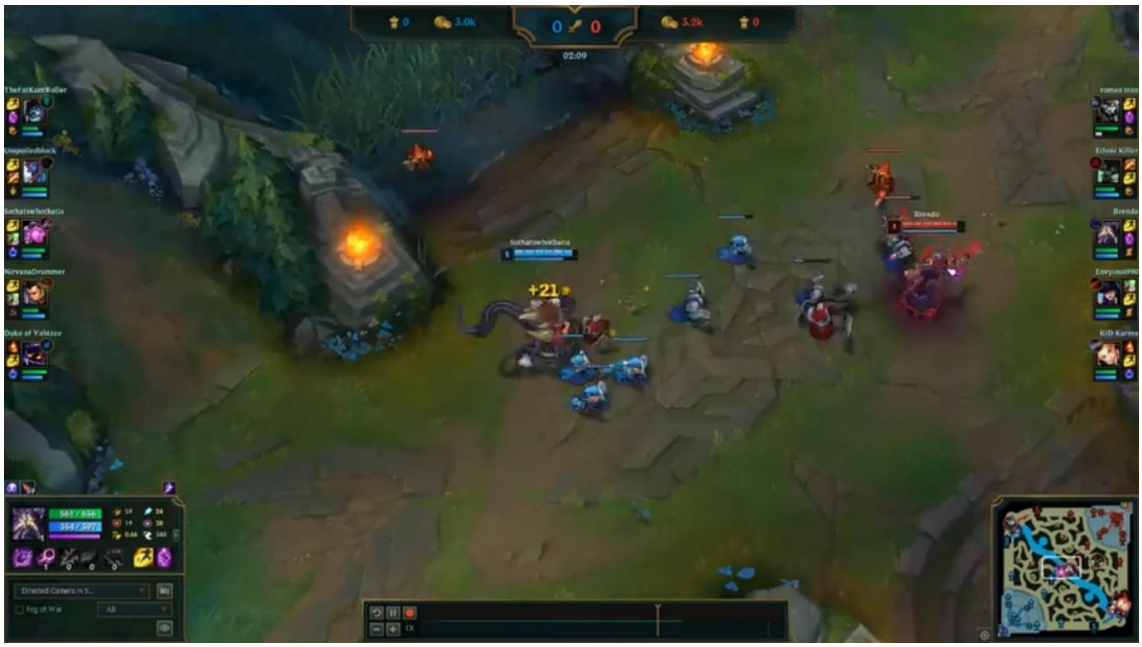

Ilustración 5 Juego League of Legends.

Es uno de los juegos más jugados actualmente, teniendo más de 100 millones de jugadores mensuales. [5]

Este juego, aparte de ser también multijugador cooperativo, aunque también se compite contra otros jugadores, se compara con el proyecto a desarrollar por la arquitectura de servidores que emplea.

Una de las claves de este videojuego son los sistemas anti trampas y la escalabilidad, permitiendo a un número muy elevado de personas competir entre sí sin miedo a las trampas.

En este juego se emplea un sistema de servidores que se van creando y destruyendo dependiendo de las partidas que se juegan, permitiendo así que cuando se necesite más potencia de cálculo, se puede abrir el servidor que alojara la partida en otro ordenador diferente por lo que no hay problemas de escalado a la hora de jugar la partida. También se emplean servidores regionales para evitar que haya demasiada latencia entre los jugadores.

A parte el servidor de cada partida es totalmente autoritativo, es decir que la partida ocurre en el servidor y los clientes funcionan enviándole acciones a realizar, evitando así que un jugador haga cosas que no están permitidas o tenga información de la partida que no podría obtener legalmente. El juego está pensado como si fuera una interfaz en la que el usuario manda una petición al servidor cada vez que interactúa con algún elemento, permitiendo así alojar toda la lógica en la nube. Si en algún momento se detecta que la acción que ha intentado hacer un jugador no es posible, simplemente no se transmite a los demás jugadores por lo que nunca habrá ocurrido.

La pega de este sistema es que las acciones no se validan hasta que el servidor las acepta, por lo que el cliente tendrá que emplear sistemas de predicción de movimiento y corrección de posición para evitar que cuando el usuario lleve a cabo una acción, haya que esperar la respuesta del servidor y que, si un usuario desincroniza su estado con el de los demás, también se emplean sistemas para volver a sincronizar los estados. [6]

3.2 Sistema para el online

Debido a que el proyecto a desarrollar será cooperativo en línea, necesitaremos algún sistema que permita la conexión entre diferentes clientes, a continuación, se van a comparar las diferentes opciones disponibles.

Aunque Unity ya incorpora un API de alto nivel para realizar juegos multijugador, lleva obsoleto desde hace tiempo por lo que se recomienda usarlo. [7]

En la documentación de Unity, se comparan las siguientes opciones.

	Stability/ support	Ease-of-use	Perfor-mance	Scalability	Feature breadth	Cost*	Customers recommend for
MLAPI	★★★★★	★★★★☆	★★★★★	★★★★☆	★★★★★	Free	Most client-server games for up to ~64 players that want a stable breadth of mid-level features
DarkRift 2	★★★★★	★★★☆☆	★★★★★	★★★★★	★★★☆☆	$100 for source	Games with high perf/ scale requirements that want to build on a stable LL layer
Photon PUN	★★★★☆	★★★★★	★★★☆☆	★☆☆☆☆	★★★★☆	$0.30/PCU	Simple and small (2–8 players) mesh-topology games
Photon Quantum 2.0	★★★★☆	★★★★☆	★★★★★	★★☆☆☆	★★★★☆	$1000/mo + $0.50/PCU	Games desiring deterministic roll-back, like MOBA games, for up to 32 players
Mirror	★★★★★	★★★★★	★★★☆☆	★★★★☆	★★★★☆	Free	Stable and proven client-server solution, loved best for its community and ease-of-use

* Note that Photon pricing provides access to the networking libraries and services, whereas other solutions are standalone networking libraries, and the cost of services is separate.

Ilustración 6 Comparación de las diferentes API

Las opciones gratuitas, MLAPI y Mirror son proyectos que surgieron a partir del antiguo sistema multijugador de Unity, usándolo como base, al ser marcado como obsoleto, por lo que aun que se ha conseguido arreglar varios fallos que tenía el mismo, siguen empleando su arquitectura, lo que limita su evolución y el arreglo de varios fallos propios de la misma. Estas APIs están pensadas para ser simples y accesibles, pero a costa de rendimiento.

A su vez, su sistema de conexión en el momento de la creación era P2P, con el tiempo se ha ido adaptando al modelo servidor-cliente debido a que es el más usado hoy en día,

aunque al no estar desarrollado inicialmente para tener esa arquitectura, siguen presentes problemas de rendimiento. Por lo que si se necesita un servidor autoritativo no son una opción a recomendar. [8]

A parte, si no tuviéramos el servidor aparecería un nuevo problema de conexión. Si los clientes no tienen correctamente configurado el rúter, como por ejemplo tener los puertos que se usen cerrados, no será posible establecer un enlace entre ellos. A su vez, si un usuario tiene peor conexión, perjudicara a todos los que estén en la misma partida.

Las demás opciones disponibles son de pago, no solo pago único, sino que en su mayoría también cobran por cada usuario mensual, por lo que haría que fuera más difícilmente escalable al aumentar los costes cada vez que aumente el número de usuarios.

Otro problema propio de las soluciones de pago es que obligan a usar sus propios servidores para alojar el nuestro, lo que provoca que no tengamos control de los mismos, impidiéndose así contratar el proveedor que se desee además de limitar donde se sitúan los mismos, perjudicando la latencia de los jugadores por no poder tener un servidor por cada región.

Debido a las razones expuestas y a la necesidad de tener el control total sobre los servidores, no solo en el alojamiento, sino a la necesidad de poder resolver cualquier incidencia o error sin depender de terceros que actualicen la API empleada, aparte de poder adaptarla completamente a las necesidades del proyecto, se ha decidido usar una API de bajo nivel que se encargue únicamente del transporte de paquetes, permitiendo así montar fácilmente toda la arquitectura, dando también la opción de cambiar fácilmente a otro sistema de transporte de paquetes si fuera necesario.

4. Planificación

Debido a que es un proyecto llevado a cabo por tres personas, la organización es muy importante. Se decidió llevar a cabo el proyecto mediante la metodología Scrum.

Se ha dividido en 3 Sprints;

El primero consistirá en la creación del escenario, los personajes y crear un prototipo jugable. Esta memoria en concreto se centrará en crear la arquitectura que en un futuro permitirá conectar a los jugadores entre sí.

El segundo consistiría en implementar más enemigos, añadir diversa funcionalidad a los jugadores y crear las mecánicas propias del escenario. En esta memoria se verá la parte relativa a la sincronización de los jugadores y los eventos, para que ambos tengan el mismo estado y puedan variar de manera imperceptible para el usuario sin que se noten los problemas típicos de la red, como perdida de paquetes o alta latencia.

El tercer sprint consistirá en la implementación de los enemigos, junto al sistema de inteligencia artificial encargado de moverlos y disparar, a su vez se harán varias pruebas de integración con todos los módulos desarrollados hasta ahora y se arreglaran o cambiaran las partes que no pasen los test.

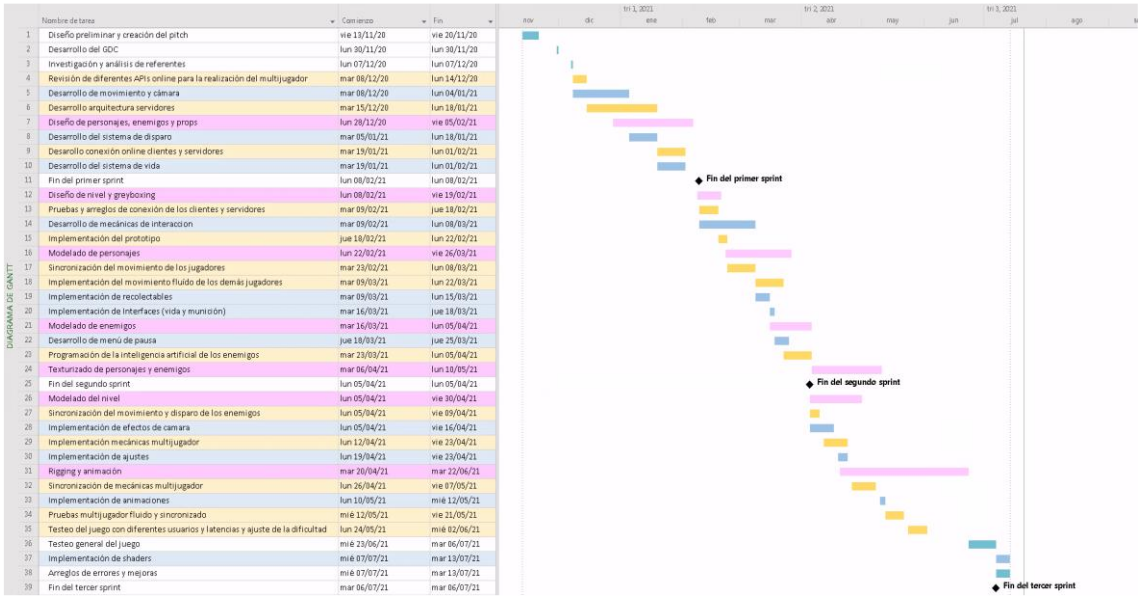

Ilustración 7 Diagrama de Gantt de los sprint y las tareas asociadas.

En la Ilustración 7 se pueden ver todas las tareas realizadas en el proyecto. Cada color indica las tareas de un miembro del equipo, siendo las tareas amarillas las relacionadas con este TFG. Se puede consultar completo en el anexo 1.

Para llevar a cabo todo lo descrito, se ha usado la plataforma Trello, que consiste en un canvas con las tareas, para ir gestionando el estado de cada una y tener una visión global del proyecto y saber cuándo un integrante del equipo ha hecho una tarea porque algunas dependen de otras por lo que es necesario acabarlas antes.

También se han llevado a cabo reuniones semanales donde se evaluaba el estado actual, se compraba si se iban cumpliendo lo especificado en los Sprints y se debatían las cuestiones que iban surgiendo propias del desarrollo.

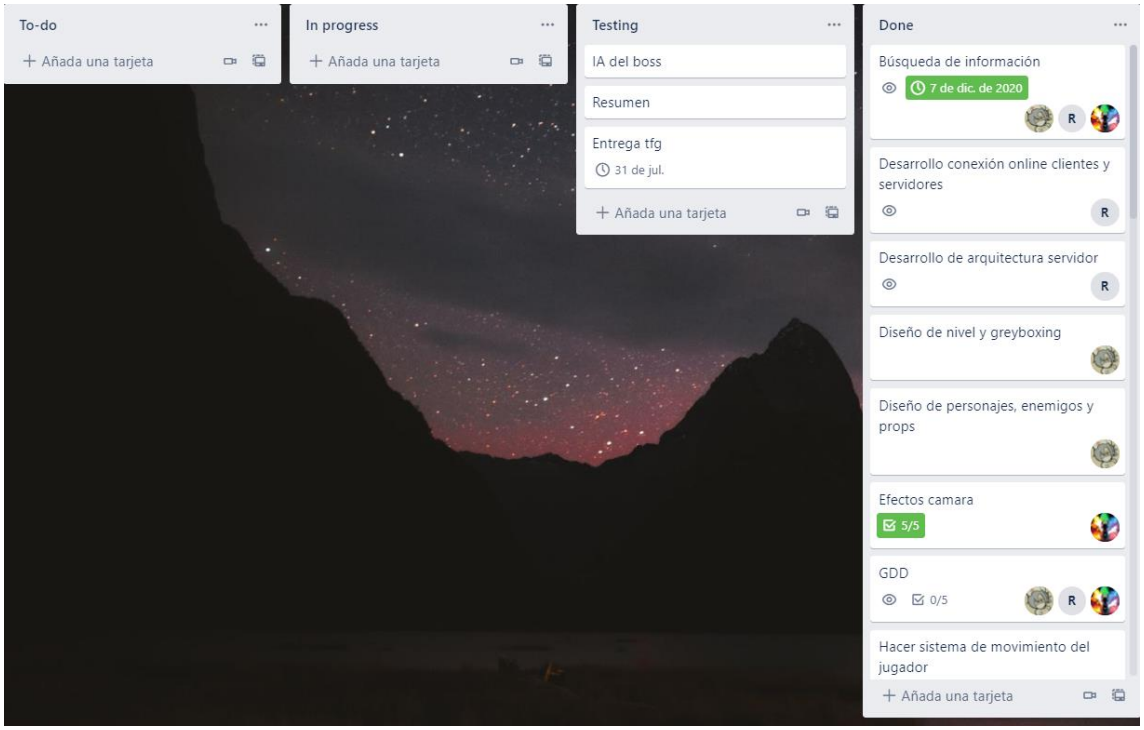

Ilustración 8 Estado final del canvas con todas las tareas que se han ido realizando.

En cuanto a tecnologías, como ya se ha comentado, se empleará Unity y c#, siendo el entorno de desarrollo Visual Studio.

Como habrá dos personas trabajando sobre el mismo código, se usará Unity Collab para compartir el código, siendo este un plugin de Unity que permite fácilmente descargarse los cambios efectuados por los demás miembros. Cuando se dieron problemas porque varias personas modificaron el mismo documentó, Unity Collab no funcionaba correctamente por lo que se empleó con Diff, un programa pensado para ver las diferencias entre dos archivos de texto y poder juntaros.

Finalmente, debido a le necesidad de probar con más de un cliente a la vez, o de separar el cliente y servidor, creamos un script para compilar todo a la vez y así tener por una parte los servidores listos para ser ejecutados y por otra tener un ejecutable del cliente que puede ser ejecutado simultáneamente más de una vez permitiendo probar lo que verían los usuarios al jugar. Se puede ver en el anexo 3.

5. Análisis y diseño

En este apartado se va a explicar de forma simple la arquitectura que se va a implementar y por qué se elige un sistema de transporte de paquetes concreto. Más información sobre la arquitectura en el apartado 6 Implementación.

También se definirán diversas mecánicas del juego que se desean desarrollar.

5.1 Arquitectura y transmisión de paquetes

Como hemos podido observar en el apartado de Estado del arte, no hay ninguna API de alto nivel que encaje completamente con el juego, las gratuitas no están pensadas para tener una arquitectura escalable o tener un servidor autoritativo. A su vez ofrecen mucha más funcionalidad que no se va a utilizar y aparte de reducir el rendimiento puede dar lugar a errores. [9]

Las soluciones de pago suelen limitar su uso a sus propios servidores, es decir que no se puede alojar donde se quiera, lo que por una parte obliga a pagar por cada usuario que se conecte a sus servidores, y por otra imposibilita dedicar un servidor a cada región para bajar la latencia global.

Por lo que las opciones a considerar serian, montar un servidor con sockets propios de c# o usar alguna API de bajo nivel que se encargue el transporte de paquetes

Se decide usar una API ya existente que se encargue de enviar y recibir paquetes porque las ventajas obtenidas de crear toda la capa de transporte serian mínimas, en cambio el desarrollo ampliaría mucho el tiempo del proyecto.

Dentro de las APIs a alto y bajo nivel, se decide usar una de bajo nivel porque para el proyecto solo se necesita un mecanismo para transferir datos, obteniendo así una API más simple de usar y sin funciones innecesarias.

Debido a que uno de los objetivos es que el juego vaya fluido con un gran número de personas y sea fácilmente escalable, buscaremos una API con buen rendimiento sobre la que montar nuestra propia arquitectura, que permita escalar fácilmente el servidor en caso de ser necesario y que sea de código abierto por si es necesario hacer alguna modificación en el futuro

Debido a todo esto, se ha decidido usar LiteNetLib porque es la que menos CPU consume, siendo esto primordial a la hora de poder dar servicio a muchos jugadores simultáneos, a su vez vemos que, aunque el consumo de RAM no es el menor, si está por debajo de la media, pero como está es más fácilmente ampliaba que la CPU, priorizamos al rendimiento del procesador como parámetro para elegir la API. A su vez

es de código abierto por si fuera necesario realizar algún cambió que beneficie el desarrollo. [11]

500 clients

GC mode: Server

Networking library	CPU usage (Max)	RAM usage (Max)	Bandwidth	Status
ENet 1.3.13	30%	64 MB	190 MB	Passed
UNet 1.0.0.9	34%	1.28 GB	195 MB	Passed
LiteNetLib 0.7.7.1	12%	292 MB	212 MB	Passed
Lidgren 1.7.0	60%	80 MB	210 MB	Passed
MiniUDP 0.8.5	28%	314 MB		Failed
Hazel 0.1.2	60%	342 MB	150 MB	Passed
Photon 4.0.29	14%	210 MB	236 MB	Passed
Neutrino 1.0	36%	362 MB	154 MB	Passed
DarkRift 2.1.0	28%	342 MB	150 MB	Passed

Ilustración 9 Comparación de las diferentes APIs para transportar paquetes.

La API elegida emplea UDP para enviar y recibir mensajes, en el caso de proyecto esto es necesario para transmitir los paquetes de la manera más rápida posible. Aunque no sea TPC, se incluyen una serie de canales para enviar los paquetes, mediante los cuales es simple emular la funcionalidad requerida, ya sea asegurar la llegada de los mismos o el orden. Se puede ver la implementación de esto en la sección 7.2 Transporte de paquetes. [12]

5.2 Mecánicas del juego

Se han planificado diversas mecánicas para los jugadores, enemigos y el propio nivel.

En esta memoria no se mostrará la implementación de las mecánicas como tal, sino la parte relativa al multijugador, sobre cómo deben ver los demás jugadores cada una de las acciones, como transmitir correctamente cada mecánica para que quede en el mismo estado entre todos los jugadores y como gestionar los problemas que se pueden derivar de la latencia de la red.

Para consultar todas las mercancías del proyecto, consultar la memoria paralela.

Por ello, debido a que muchas interacciones sean diferentes para el jugador, muchas son iguales en la forma de implementación. Por ejemplo, al interactuar con objeto del entorno, se transmitirán de idéntica manera, pero variando el identificador del objeto usado. Dándose el mismo caso con todo tipo de movimientos y saltos. También el disparo es independiente de cada arma, por lo que se también se gestionaran todos los disparos de la misma manera, con un paquete de disparo que varié el tipo de bala usada y la posición desde la cual se dispara.

En resumen, todas las mecánicas se han reducido a cinco que serán configurables para conseguir sincronizar todos las existentes.

Movimiento

El jugador puede moverse libremente por un mundo en tres dimensiones, a su vez también a los demás usuarios variando su posición.

Salto

Se permite saltar, lo que realizara una animación especial. Se ha diseñado como mecánica a parte del movimiento para evitar conflictos al subir pendientes muy elevadas puesto que la posición variara de manera similar.

Disparo

El jugador podrá disparar balas que dañaran a los enemigos. Dependiendo del arma que se emplee, se generaran diferentes balas.

Interacción con el entorno

Se han creado diversos elementos activables, palancas puertas y botones. El usuario podrá interactuar con ellos para abrir puertas, mover plataformas u obtener armas.

Movimiento de los enemigos

Los enemigos emplearan un sistema de búsqueda de caminos para seguir al jugador y dispararle cuando estén cerca.

6. Implementación

En este apartado se expondrá de principio a fin como se ha llevado a cabo el proyecto y los problemas que han ido surgiendo.

Esta parte de la memoria tratará los temas de multijugador y enemigos, para ver como se ha llevado a cabo la implementación de los demás elementos, consultar las memorias de los diferentes miembros del grupo.

Se dividirá en cinco secciones, arquitectura y organización del servidor, transporte de datos, sincronización de los estados de los jugadores, implementación de los enemigos e instalación del servidor en la nube.

En cada una se explicará el trabajo realizado y los problemas asociados.

6.1 Arquitectura y organización del servidor

La principal dificultad del proyecto consiste en desarrollar una arquitectura solida que permita por un lado dar servicio a un gran número de jugadores y por otro lado evitar las trampas. Después de realizar la arquitectura base e ir mejorándola conforme han aparecido necesidades nuevas o se han las pruebas, se ha llegado al siguiente diseño.

Consiste en una arquitectura cliente-servid. Un servidor maestro al que se conectaran todos los usuarios, un servidor de juego que se iniciará sólo cuando los jugadores quieran jugar y al que se conectaran, que actuara a su vez de servidor para los jugadores y de cliente con el servidor maestro, y finalmente el cliente de cada jugador.

6.1.1 Servidor maestro

El servidor maestro es el primer servidor en ser ejecutado, una vez inicializado su función consiste en esperar conexiones.

Si recibe una conexión lo primero es ver si se está conectando un jugador o un servidor de juego. Este último se inicializará al recibir una petición de jugar de los clientes por lo que al haber arrancado correctamente, se conectará al servidor maestro para informar de que se ha inicializado correctamente y comunicar datos como los puntos de cada jugador al finalizar la partida para poder crear una clasificación.

Para distinguir entre jugadores y servidores se utiliza una clave, si el cliente que intenta conectarse conoce la clave, quiere decir que es un servidor de juego, por lo que se acepta la conexión, aunque se haya alcanzado el límite de clientes conectados

En caso de ser un jugador, primero se tiene que asegurar que hay sitio para nuevos clientes, si no hubiera limite podría darse el caso de que se colapsara el servidor no permitiendo jugar a nadie, por lo que es preferible limitar la cantidad.

Con las pruebas llevadas a cabo el servidor soporta correctamente 100 clientes conectados así que se decide limitarlo a 100. En un futuro habría que revisar el límite debido a que, si se añaden diferentes funciones como un chat o una base de datos, puede aumentar la carga por jugador por lo que se reducirá el número de jugadores a los que se puede manejar correctamente.

Si no hay sitio para nuevos jugadores se le devuelve un mensaje al jugador diciendo que actualmente el servidor está lleno. En este caso el cliente intentará volver a conectarse automáticamente en 10 segundos para comprobar si ha quedado algún hueco libre.

En caso de haber sitios disponibles, el jugador se conecta correctamente y pasa a un estado de espera, en el que el cliente solo enviara un paquete conocido como “keepalive” que contiene un bit y se reenvía cada segundo, su función es determinar si la conexión sigue activa. Si el cliente no contesta a ningún paquete de los enviados en los siguientes 10 segundos se cerrará su conexión.

En cualquier momento que el cliente decida jugar una partida, pulsará el botón de jugar situado en la interfaz. En este momento se enviará un paquete indicando que se quiere iniciar una partida y el tipo de partida. Si no hay usuarios suficientes para jugar la partida solicitada, el servidor registrará la petición y añadirá al jugador a la cola de espera para jugar. En el momento que se reciba otra solicitud de juego, se volverá a comprobar si hay jugadores suficientes.

A su vez si un jugador se desconecta o envía un paquete de cancelar partida pulsando nuevamente el botón de jugar, será sacado de la cola de juego.

Si al recibir la petición de juego, contando con el jugador que ha hecho la petición hay suficientes jugadores como para empezar la partida, el servidor iniciará la creación de un proceso a parte que contendrá el servidor de juego.

En este momento se le pasará al cliente la ip y puerto del nuevo servidor de juego creado para que pueda conectarse a él.

En el caso del proyecto desarrollado, ya ip de ambos servidores es la misma porque están alojados en el mismo sitio.

En cuanto a los puertos, se va incrementando el número de puerto usado empezando desde el 8888. Cuando un servidor de juego se cierra su puerto se añade a una lista de

puertos a reutilizar por lo que, si hay alguno en esa lista, en vez de usar un puerto nuevo se reutilizara uno de la lista.

El poder conectarse a una ip diferente se ha creado con la intención de poder alojar en cualquier otro sitio los servidores de juego, permitiendo no solo que sean fácilmente paralelizadles al gestionar cada hilo un servidor, sino también poder abrir el servidor de juego en cualquier otro sitio, logrando así aligerar la carga del proveedor que aloje al servidor maestro, a su vez también se permite contratar diferentes proveedores, uno en cada región e iniciar el proceso del servidor de juego en el proveedor que tenga menos latencia con los jugadores que van a iniciar la partida.

Una vez arrancado el proceso del servidor de juego y enviada la ip y el puerto al cliente la petición ya está atendida, puesto que será el cliente el que se conecte al servidor automáticamente. Así que se vuelve al modo de espera, esperando nuevas conexiones o nuevas peticiones. [13]

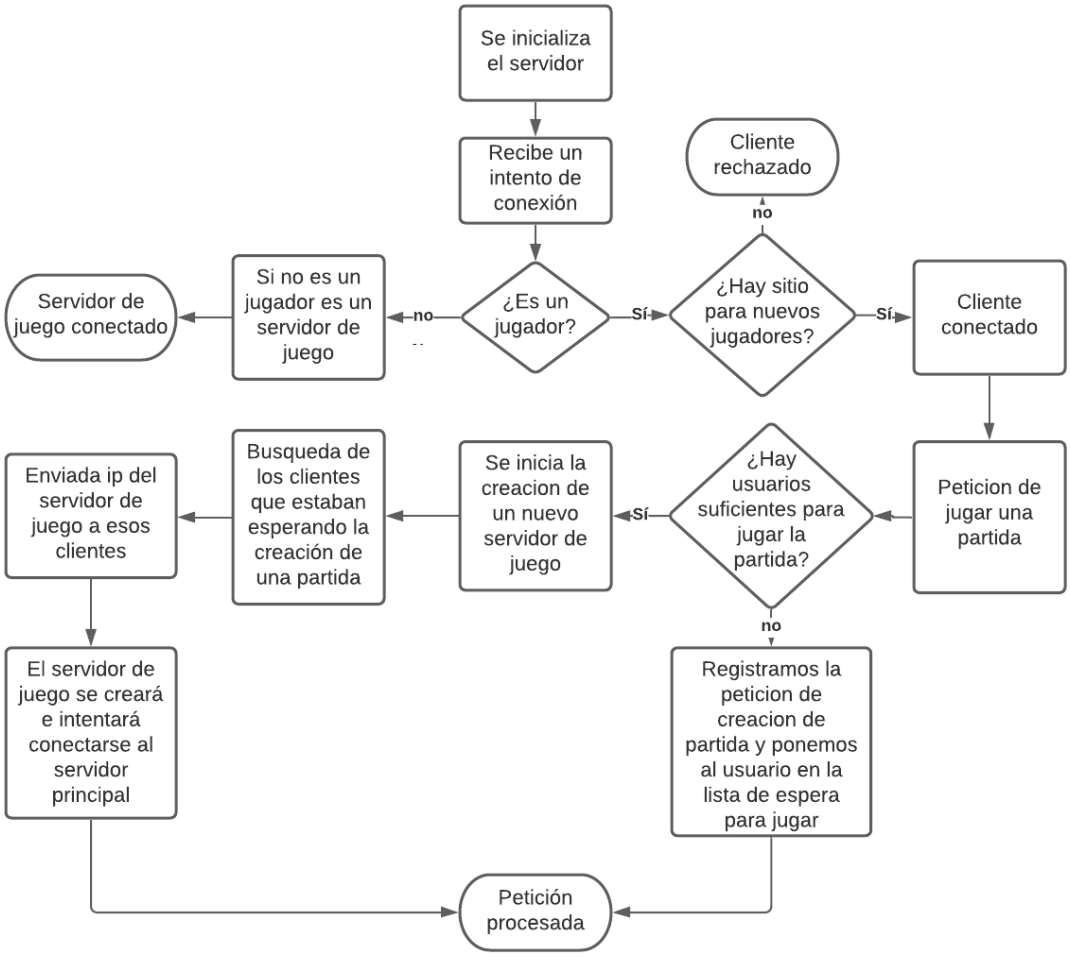

Ilustración 10 Diagrama de funcionamiento del servidor maestro.

6.1.2 **Servidor de juego**

Este servidor de juego se inicia bajo demanda, es decir que solo se arranca cuando el servidor maestro lo requiere porque algún jugador ha elegido jugar.

Al iniciarse, se intentará conectar al servidor maestro, si no lo consigue dejará un log con la causa y los jugadores cuando no consigan conectarse a la partida, volverán al servidor maestro.

Una vez conectado con el servidor maestro, se esperará a que se conecten todos los jugadores.

Una vez conectados todos, los clientes empezaran a cargar el escenario de juego. Cuando todos hayan cargado empezará la partida.

Si hay algún jugador que no consigue conectarse al servidor o no carga correctamente el escenario, la partida se cancelará, desconectando a todos los jugadores, haciendo que estos se reconecten al servidor maestro, notificando al mismo que la partida ha finalizado por lo que puede liberar su puerto y cerrándose el proceso puesto que ya no existe la partida.

En caso de conectarse y cargar todos correctamente, la partida se iniciará y transcurrirá con normalidad, encargándose el servidor de validar y transmitir todos los datos entre los jugadores. Cuando el juego acabe el servidor realizara el mismo proceso que cuando no han podido conectarse todos, desconectando a los jugadores, notificando al servidor maestro que la partida ha acabado y cerrando el proceso. En este caso se pueden añadir mensajes entre el servidor de juego y el servidor maestro, como el resultado de la partida para poder crear un sistema de clasificación de jugadores.

A parte, si durante el transcurso de la partida un jugador se desconecta, el servidor dará la opción a los jugadores restantes de terminar la partida o de seguir jugando sin el jugador desconectado.

En caso de algún error critico en el servidor de juego, este registrará el error para una futura revisión y se cerrará, por lo que pasados diez segundos sin haber contestado al paquete de keepalive, los clientes volverán al servidor maestro.

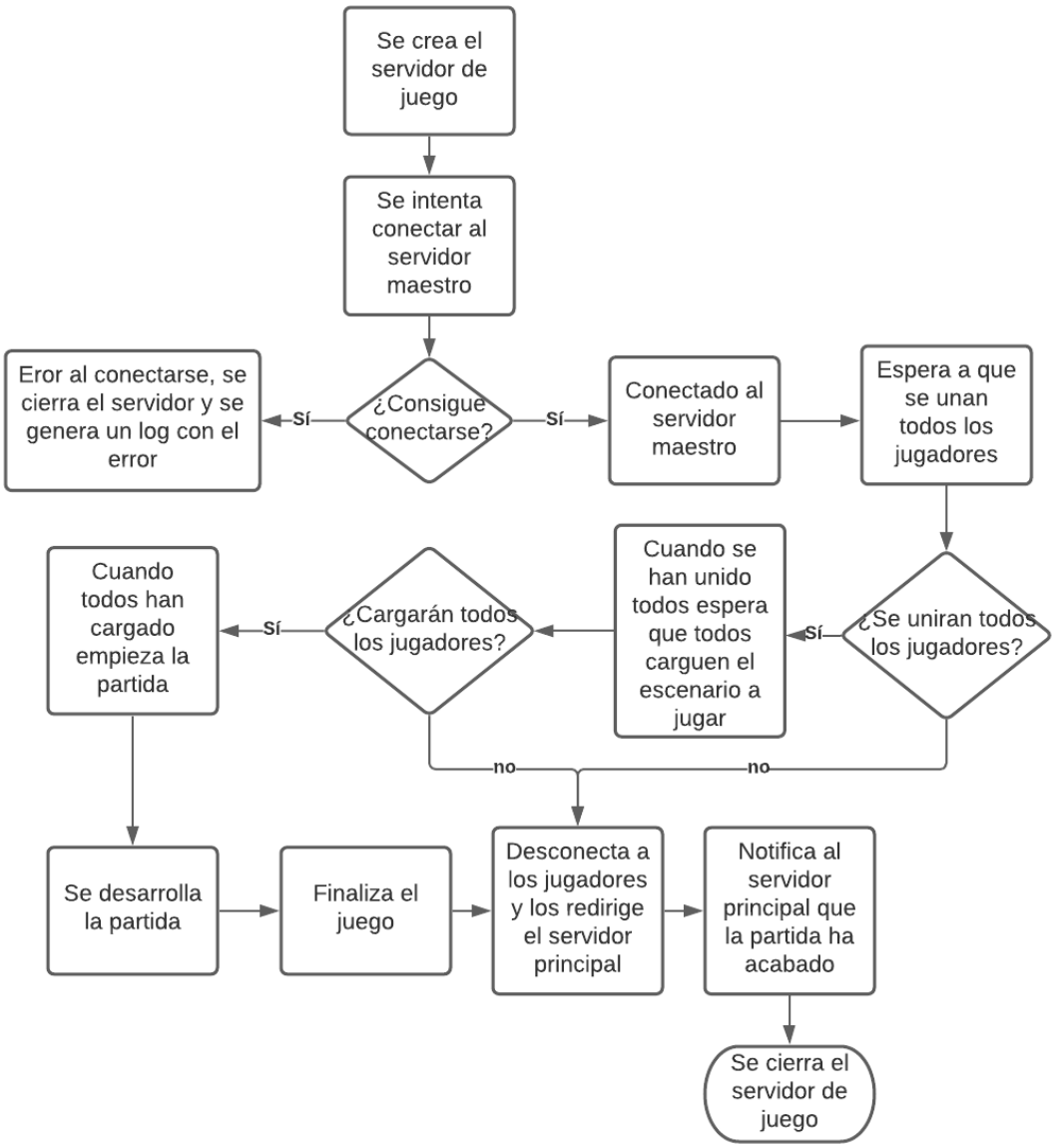

Ilustración 11 Diagrama de funcionamiento del servidor de juego.

6.1.3 **Cliente**

Al iniciar el juego, automáticamente el cliente se conectará al servidor maestro.

Si el servidor no admite más jugadores, el cliente volverá a intentar conectarse en 10 para ver si se ha quedado algún hueco libre.

En caso de que la conexión sea satisfactoria el cliente se quedara en modo espera, enviando cada segundo el paquete “keepalive” que permitirá determinar que la conexión sigue activa. Si no se recibe respuesta en los próximos 10 segundos, la conexión se cerrará y el cliente volverá a intentar conectarse.

Una vez el jugador quiera empezar una partida y pulse el botón para jugar se enviará una petición al servidor para encontrar una partida.

Si no hay ningún otro jugador esperando para jugar una partida con las opciones que haya elegido el cliente, el servidor añadirá al usuario a una cola a la espera de encontrar otro jugador. En el momento que otro cliente decida jugar, ambos se conectaran al servidor de juego creado para su partida.

Si ya había otro jugador esperando para jugar, el cliente se unirá junto al otro jugador al servidor de juego.

Durante la partida ambos clientes enviaran y recibirán datos como su posición o los disparos que efectúen.

Finalmente, cuando finalice el juego, ambos se conectarán de vuelta al servidor maestro a la espera de volver a realizar alguna otra acción.

A parte del flujo mencionado, si el cliente en algún momento se desconecta del servidor por algún fallo de internet o por algún error en el servidor de juego, automáticamente intentara reconectarse al servidor maestro logrando así recuperarse de un fallo.

Cabe remarcar que la clase cliente emplea un patrón singleton, esto quiere decir que hay un único punto de entrada para acceder a toda la gestión de envío y recepción de paquetes. Cualquier cosa que desee ser enviada o recibida deberá ser mediante esta clase, teniendo una variable estática que contiene la instancia a la que todas las clases pueden acceder. Si en un futuro se desea cambiar el sistema de envío de paquetes, bastaría con cambiar esta clase.

La clase mencionada se inicializa en la pantalla de carga y no se destruye nunca, ni en el cambio de escenas puesto que entonces se perdería la conexión.

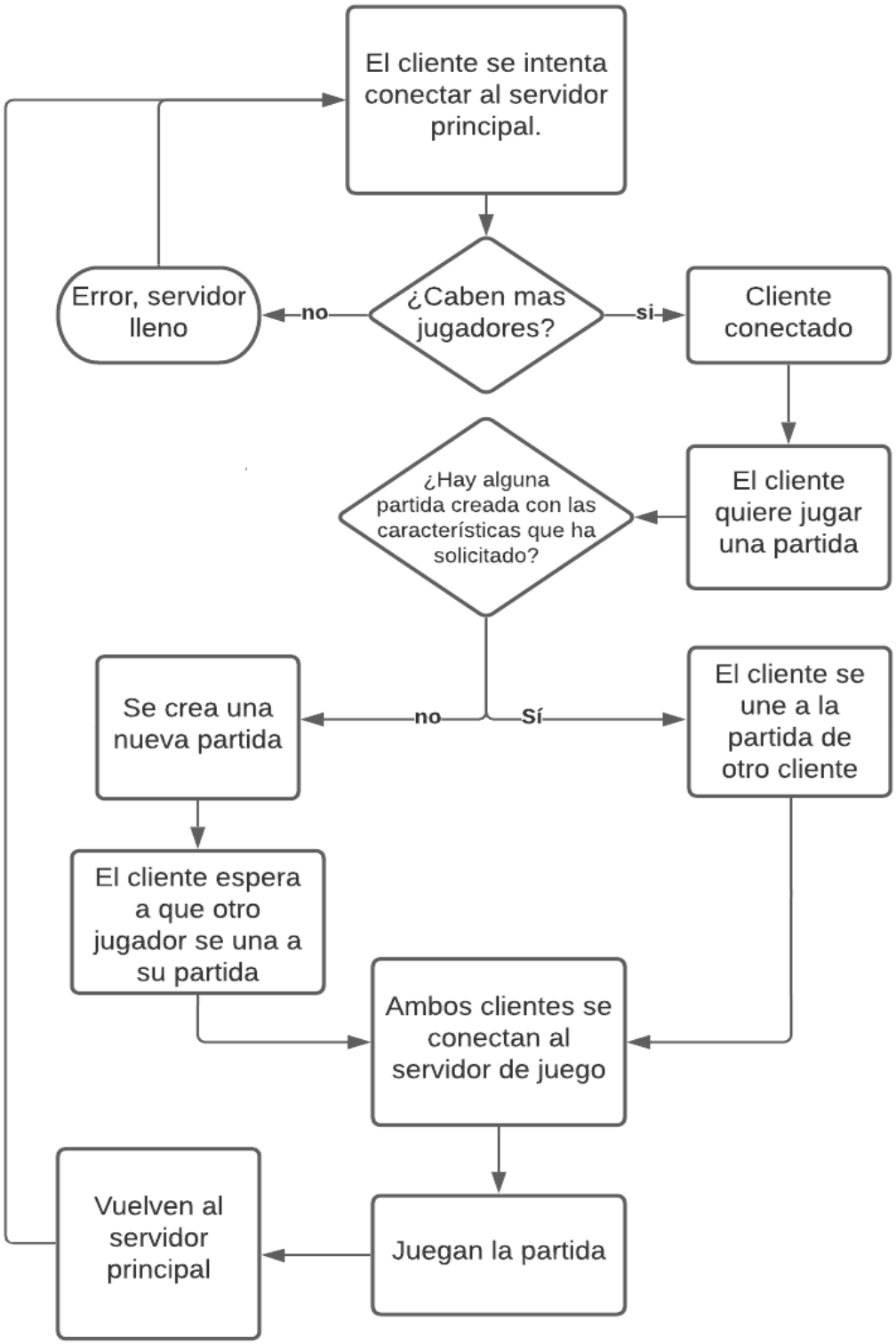

Ilustración 12 Diagrama de funcionamiento del cliente.

6.1.4 Parámetros de configuración

A parte de la arquitectura explicada anteriormente, cabe mencionar algunos parámetros importantes que influyen en el comportamiento del sistema.

En el anexo se puede ver la declaración en el código de cada uno de ellos.

Latencia máxima: 300 milisegundos

Latencia máxima permitida para los clientes, como se ha podido comprobar al realizar diversas pruebas, si es muy elevada afecta a la experiencia de juego. Actualmente se emplea para avisar al cliente de que puede haber algún problema con su conexión, pero en un futuro puede usarse para impedir que jugadores con mala conexión entorpezcan la experiencia de los demás.

Tiempo sin respuesta para desconectarse: 10 segundos

Cada segundo se envía un paquete llamado "keepalive" cuya única función es comprobar si la conexión sigue activa, si tras el periodo indicado por este parámetro no se ha recibido respuesta, la conexión se dará por finalizada.

Clientes máximos: 100 clientes

Número máximo de clientes simultáneos conectados al servidor maestro. Se han realizado pruebas y el servidor soporta correctamente 100 clientes. Si en un futuro se añade más funcionalidad habría que revisar este parámetro al ser posible que se consuman más recursos por cliente.

Tiempo de espera de conexión de los clientes a la partida: 10 segundos

Tiempo máximo de espera para que los jugadores se conecten al servidor de juego. En caso de superar este tiempo y no estar todos conectados se cancelará la partida.

Tiempo de carga de los clientes en la partida: 10 segundos

Tiempo máximo de espera para que los jugadores carguen la partida una vez se hayan conectado al servidor de juego. Podrían conectarse correctamente, pero cerrar el juego por lo que no cargarían. En caso de superar este tiempo y no estar todos conectados se cancelará la partida.

Tiempo para reintentar la conexión fallida: 10 segundos

Al intentar conectarse al servidor y producirse algún tipo de fallo, el cliente se esperará el tiempo definido aquí para volver a intentar reconectarse y así no saturar el servidor con intentos de conexión.

6.2 Transporte de datos

Para que la arquitectura definida anteriormente es importante disponer de un sistema de intercambio de información. En esta sección se verá cómo se implementa la API escogida (LiteNetLib) y como se prepara para adaptarla a las necesidades del proyecto.

La API elegida proporciona un mecanismo para enviar paquetes que consiste en un método "send" al cual se le pueden pasar como parámetros los datos a enviar. Al recibirlos se activará un evento de recepción de paquetes que contendrá los datos enviados.

6.2.1 Arquitectura para enviar y recibir paquetes

En este caso se ha preferido añadir una capa extra de abstracción para poder manejar más fácilmente los paquetes, pudiendo reutilizarlos y sabiendo así que tipo de paquete se ha enviado cada vez.

Se decide crear una clase llamada "PacketList" que contendrá todos los paquetes y lo que contiene cada uno. En el anexo se puede consultar el código de la clase.

El objetivo es crear un paquete por cada acción diferente que se quiera enviar, logrando así saber que acción solicita el cliente al recibir el paquete y mantener un único punto donde se declaran los paquetes. En la clase nombrada anteriormente estarán declarados todos, incluyendo los datos que emplee cada uno de ellos.

Un ejemplo sería el paquete "PositionPacket" que contendrá la posición y la rotación del jugador, por lo que para enviar la posición de un jugador solo habría que crear un nuevo paquete de este tipo y darles valor a sus parámetros de posición y rotación. Para más ejemplos ver el anexo.

Para recibir los paquetes es necesario declarar un listener con el tipo de paquete a recibir y el código a ejecutar en ese caso. Por lo que al recibir un paquete se activará automáticamente el código asociado, evitando teniendo que comprobar cada vez que tipo de dato estamos recibiendo. Así mismo, algunos paquetes no es necesario emplearlos en ambos servidores, por ejemplo, un paquete que determine la posición de un jugador solo será usado por el cliente y el servidor de juego por lo que no hará falta declarar su recepción en el servidor maestro.

La pega de esta arquitectura es que se añade un "overhead" de 4 bytes a cada paquete para permitir discernir automáticamente lo que estamos enviando. Se prefiere este sistema porque facilita enormemente el desarrollo y la organización de todo el proyecto, además como se ha podido comprobar en el apartado de pruebas, la cantidad de datos transmitidos no es muy elevada y es fácilmente gestionable por el servidor.

Finalmente, también se crea un mecanismo para poder enviar tipos de datos de más complejos consiguiendo que se serialicen automáticamente. Para ellos será necesario al crear el paquete implementar la clase ”INetSerializable” y sus métodos. Con esto se consigue enviar fácilmente el estado de un enemigo, transmitiéndose de una vez todos los datos que contiene como la posición, vida o tipo de enemigo. Se puede consultar el código completo en el anexo.

6.2.2 Tipos de envío de paquetes

La API encargada del transporte de paquetes emplea UDP, como el proyecto a desarrollar es un videojuego es preferible que sea así al tener más rendimiento que TPC.

Aunque al ser UDP carezca del mecanismo de comprobación de recepción de paquetes de TPC, se permite configurar el envío de cada paquete pasándole al método “send” un parámetro adicional que consiste en el tipo de canal por el cual se va a enviar el mismo, habiendo canales que replican las comprobaciones propias de TPC y permiten usarlas en caso de ser necesario. Si no se añade ningún parámetro extra, el envío funcionara como UDP, pudiendo no llegar los paquetes o hacerlo de manera desordenada.

A continuación, se repasarán los empleados.

Reliable ordered

Los paquetes tendrán confirmación de llegada, si estos no llegan en un intervalo de tiempo definido se volverán a enviar. Además, se comprobará que lleguen en el mismo orden de emisión y que no llegue un paquete duplicado.

Este tipo de envío es usado para eventos críticos que deban asegurar su llegada en un orden determinado y no puedan llegar dos veces, como conectarse a un servidor y luego cargar la partida.

También se emplea para la activación de objetos en el escenario porque estos paquetes no pueden perderse o la acción no se realizaría, además el orden en el que se realizan también importa.

Reliable sequenced

Los paquetes llegaran ordenados. No se asegura la llegada de todos ellos. Se asegura la llegada del ultimo, es útil para actualizar el estado de algo que no envíe muchos paquetes.

Se emplea para transmitir el salto porque al realizar esta acción se envía un solo paquete que no debe perderse y en caso de acumularse saltos, solo se deberá recibir el ultimo.

Reliable unordered

Los paquetes tendrán asegurada su llegada, pero pueden llegar desordenados o duplicados.

Se emplea para el disparo puesto que un disparo no puede perderse, pero no importa el orden en el que lleguen.

Sequenced

Los paquetes llegaran en orden, pero no se asegura su llegada.

Esto se emplea para enviar la posición, debido a que si un paquete se pierde no es un gran problema puesto que se envían más. Igualmente, si llega un paquete anterior será desechado puesto que esa posición será vieja.

Unreliable

No se asegura nada, los paquetes pueden perderse o llegar desordenados. [14]

6.3 Sincronización de los estados de los jugadores

Una vez ya se ha definido la arquitectura del sistema y un método para enviar y recibir datos, el siguiente paso es conectar dos clientes a la misma partida y que las acciones de cada jugador puedan ser vistas por el otro.

Todos los eventos propios del servidor maestro como conexiones o creación de partida son del tipo autentificativo porque es el servidor el que gestiona y posee todos los datos.

El tipo de acciones que puede realizar cada jugador dentro de la partida, y que por lo tanto habrá que enviar al otro cuando se produzcan, se ha reducido a cuatro.

Acciones autentificativas	Activación de elementos
Acciones reenviadas	Futuro chat
Acciones validadas	Disparo del jugador Movimiento del jugador Salto del jugador

Ilustración 13 Tipos de paquetes enviados durante la partida

La manera que tiene el servidor de gestionar los eventos relativos a la partida se puede clasificar en tres tipos, dependiendo del tipo de validación que se haga sobre el mismo en cuanto a si contiene datos correctos.

A continuación, se verán las diferencias entre cada tipo y en cual están incluidos los tipos de acciones a sincronizar mencionados anteriormente.

6.3.1 Comportamiento autoritativo

Este tipo de servidores se denomina así porque la autoridad recae sobre el mismo.

Un servidor autoritativo en lo referente juegos multijugador, consiste en trasladar el control total de la partida al propio servidor y hacer que los jugadores actúen como espectadores que pueden mandar comandos para influir en la misma.

En este caso, si un cliente quisiera variar su posición en vez de enviar la posición donde va a estar, enviaría la dirección a la que se quiere mover y el servidor haría los cálculos para saber dónde debería estar ese jugador al realizar esa acción. Devolviéndole este resultado al propio jugador para que actualice su nueva posición. Impidiendo así cualquier tipo de manipulación de los estados que vaya contra la lógica programada en el servidor.

Por ejemplo, un cliente malicioso podría intentar no morir al recibir un disparo, pero dado que la partida se desarrolla en el servidor, en el momento que una bala impacte en él, todos los demás clientes verán que ha muerto, aunque el intente lo contrario, debido a que no tiene control sobre todos los aspectos del juego.

Aunque este sistema puede proteger un juego complemente ante las trampas, también tiene principalmente dos problemas que veremos a continuación.

Estados retrasados con respecto al servidor

Debido a que la partida está realmente en el servidor, desde que realizamos una acción hasta que realmente se ejecuta, devuelve el resultado y el cliente lo aplica pasa un tiempo, que, aunque sea pequeño entorpece enormemente la experiencia.

En un caso normal, un cliente puede tener 100 milisegundos de latencia, por lo que, sin contar ningún otro factor, se emplearían 200 milisegundos en enviar una acción y obtener el resultado, lo que hace parecer que el videojuego se ejecuta con retraso.

Por ello es necesario incluir sistemas de predicción, que se encarguen de realizar la acción localmente a la vez que se envía al servidor. Aunque esta solución también implica algunos problemas, se pueden resolver de manera satisfactoria.

Por un lado, si el sistema no es determinista, esto quiere decir que con los mismos datos de entrada siempre devuelva la misma salida, cosa que ocurre en Unity debido a la representación inexacta de los números decimales, puede ocurrir que nuestro sistema de predicción suponga un estado que finalmente no se cumpla, lo que llevará a tener que implementar también un mecanismo de reconciliación de estados, para poder pasar de la

manera menos molesta para el usuario del estado predicho al devuelto por el servidor. Aunque las diferencias iniciales entre estados sean pequeñas, si no se corrigen rápidamente dan lugar a comportamientos totalmente diferentes por lo que hay que buscar un equilibro para no estar revisando constantemente si un estado es válido, lo que implicaría problemas de rendimiento y revisarlos lo suficientemente frecuente como para poder arreglarlos sin que el usuario lo note porque aún no se ha desviado mucho uno del otro.

Otro problema común es la realización de una acción, como coger un objeto, que en realidad ya no se puede realizar porque en el servidor ese objeto ha desaparecido, aunque aún no se haya llegado la notificación, ya sea por mera casualidad o por un retraso en la conexión de internet.

Este caso puede ser también resuelto por el sistema de reconciliación de estados mencionado anteriormente.

Aunque existe un mecanismo para resolver más eficientemente los conflictos en los que un jugador actúa sobre un elemento que ya no está, consistiendo que retroceder en el tiempo hasta que el usuario ejecutó esa acción y volver a realizar todas las acciones partiendo de ese cambio, esto queda fuera del alcance de este proyecto debido a que no hay acciones que puedan afectar a un mismo elemento y producir estados opuestos. Por ejemplo, si ambos dispararan al mismo enemigo a la vez, este simplemente moriría sin importar que jugador ha ejecutado la acción antes.

También, hay que hacer hincapié en recrear correctamente la partida entre el cliente y el servidor, porque si por algún problema de diseño hay algún elemento diferente, como una pared en una posición ligeramente distinta del servidor, llevara a que las predicciones fallen constantemente y el cliente vea cambios constantes en su posición.

Incremento de los cálculos realizados en el servidor

Como ahora el servidor es el encargado de todo, tiene que procesar todos los datos referentes a la partida, por lo que no solo es transmitir y enviar paquetes sino calcular como afectará al estado los nuevos datos incluidos, además de todas las simulaciones físicas que suelen ser bastante costosas. Haciéndolo difícilmente escalable en número de jugadores simultáneos por partida y aumentando los costes para alojar un servidor.

La única solución a este problema, es cambiar la funcionalidad de autoritativo con el control de la partida a actuar como comprobador de los datos introducidos por cada cliente.

Como se puede ver no existe una solución óptima y habrá que elegirla en función de las necesidades de cada proyecto, en este proyecto se ha elegido tener la funcionalidad autoritativa para datos con más importancia como el disparo de cada jugador y una función de comprobación de datos introducidos para la posición, este último se detallara en la sección de comportamiento como comprobador de paquetes.

Implementación del comportamiento autoritativo en el juego

Después de conocer los problemas que puede aportar este tipo de comportamiento autoritativo, se procede a mostrar la solución implementada con este tipo de sistema.

Cualquier método no relativo a la partida se podría incluir aquí porque es el propio servidor el que se encarga de leer los datos y realizar la operación que considere con ellos, como eventos de conexiones o solitudes de iniciar una partida. Estos no tienen mayor complicación por lo que el cliente no espera una respuesta inmediata al realizar estas acciones, la partida se notificará cuando esté lista y no se puede prever cuando será, así que simplemente el cliente envía las peticiones y el servidor contestara cuando las haya procesado.

Como se ha mostrado en la ilustración 13 el único paquete usado de tipo autoritativo en la propia partida es el de la activación de elementos.

Esto se ha decidido así porque activar elementos del escenario como puede ser una palanca o un botón es básico para poder avanzar en el nivel por lo que no se puede permitir que se hagan trampas con ella, a parte el hecho de gestionar cuando se puede disparar no añade mucha sobrecarga al servidor.

Hay definidos tres tipos de elementos usables en el escenario, palancas, botones y puertas. Al pulsar sobre cualquiera de ellos se produce una acción determinada.

En todos los casos se gestiona de la misma manera, el cliente envía un paquete con el identificador del objeto a activar mientras localmente ejecuta ya la acción resultada de su activación. A su vez el servidor comprueba que la posición del jugador sea lo suficientemente cercana a la de ese elemento y que realmente ese cliente puede activarlo, puesto que hay elementos que no son usables por todos los jugadores.

En caso de superar las comprobaciones, el servidor enviará un paquete a todos los clientes donde se indicará el elemento que ha sido activado.

En caso de que el servidor no pueda ejecutar la activación del elemento debido a que el jugador no está lo suficientemente cerca o no tiene permisos, no se reenviara ningún paquete por lo que nadie vera el resultado de una acción fraudulenta.

Por lo que la única manera de ejecutar estas acciones será la manera pensada por el programador, impidiendo cualquier otro tipo de manipulación intencionada. [15]

6.3.2 Comportamiento reenviador de paquetes

Este tipo de servidor consiste en reenviar los paquetes de un usuario a los demás. Su única función es permitir el intercambio de datos entre ellos, haciendo de punto de unión.

Las principales ventajas son, por una parte, que el gasto de procesamiento es mínimo puesto que no comprueba nada, la lógica la lleva cada cliente así que en caso de necesitar escalar el número de usuarios o de funcionalidades se podrá hacer fácilmente.

Por otra parte, al tener un servidor al que conectarse, se evitarán problemas de conexión típicos de la arquitectura P2P, para conectar clientes entre ellos hace falta que alguno tenga los puertos abiertos y actúe de servidor, o algún tipo de sistema de redirección de puertos que no siempre funciona por lo que, al haber un servidor entre ellos, se evitaran todos los problemas de este tipo. Igualmente, si un cliente tiene mala conexión no afecta al rendimiento de los demás puesto que solo verán con retraso a ese jugador. En caso de conexión cliente a cliente, ese mismo podría ralentizar todo el sistema.

La desventaja principal es que, mediante este método, no se validan los datos de ningún paquete, simplemente los reenvía a los usuarios indicados por el emisor, por lo que no protege contra ningún tipo de trampas ni datos incorrectos generados por algún fallo del juego.

Este tipo de funcionamiento actualmente no se emplea actualmente en ningún método del servidor, se menciona porque un futuro proyecto será un chat integrado, en este caso se emplearía esta funcionalidad puesto que simplemente tendría que reenviar el texto de cada jugador.

6.3.3 Comportamiento como comprobador de paquetes

Este sistema une partes de los dos mencionados anteriormente.

Cada cliente gestiona la partida localmente y es el servidor se encarga de comprobar los mensajes enviados de cada jugador para ver si son válidos, retransmitiéndolos a los jugadores pertinentes. En caso de no pasar la comprobación, estos se desecharán por lo que si ha habido un intento malicioso este no se verá reflejado.

A diferencia del servidor de reenvío de paquetes, este sí que comprueba los datos enviados por cada cliente, por lo que puede parar los intentos de hacer trampa. Aunque, por el contrario, necesita más de tiempo de computación para poder validar los datos de cada paquete.

Por otra parte, al contrario que el servidor autoritativo, éste no es el encargado de gestionar toda la partida por lo que, al validar los datos, es necesario dejar un margen para comprobar si un paquete es válido, por lo que un cliente malicioso puede usar ese trozo variable para ganar una ligera ventaja. Por ejemplo, al enviar un cliente la nueva posición, el servidor calcula un intervalo de posición en el que podría estar, no puede ser exacto debido a variables de la red como la latencia, si la posición enviada está en ese rango se transmite a los demás clientes.

La principal ventaja comparado con el servidor a autoritativo consiste en trasladar la mayoría de los cálculos al cliente, logrando así reducir enormemente la carga del servidor, pudiendo dar servicio a muchos jugadores y ahorrando costes de alojamiento.

La mayoría de la funcionalidad empleada en el proyecto pertenece a este tipo. Ofrece suficiente protección ante clientes ambiciosos mientras permite varios jugadores simultáneos sin un alto coste. A su vez facilita el coste de desarrollo y mantenimiento, debido a que de esta manera datos como la posición de los jugadores no se guardan en el servidor por lo que se evita tener que duplicar la lógica de movimiento y conseguir que sea consecuente entre el cliente y el servidor. Mediante un buen ajuste de los métodos para validar los datos se ha conseguido un comportamiento muy similar a un servidor autoritativo sin tener los problemas asociados a este.

Implementación del comportamiento comprobador de paquetes en el juego

En esta sección se describirá no solo la implementación de este tipo de sistema sino también los problemas que surgen derivados de su implantación y como han sido resueltos.

Se distinguirán los siguientes.

Disparo

Cada jugador enviará un mensaje al servidor con la posición desde la cual desea disparar. El servidor comprobará primero que la posición recibida concuerde la última posición conocida del jugador, dependiendo del transcurrido desde el último mensaje donde se daba a conocer la posición y la velocidad a la que se mueven los jugadores, se puede establecer un rango de posiciones desde la cual podría estar disparando el jugador. A parte se deja un pequeño margen de diferencia en las posiciones por si ha habido alguna anomalía en la red y los paquetes no llegan de forma constante.

Si esta comprobación es válida, se revisa el tiempo en el que ejecuto el ultimo disparo ese jugador, si ha pasado el tiempo requerido entre disparos, se reenvía este paquete a todos los demás clientes haciendo que vean el disparo efectuado.

En este caso, en cuanto el cliente envía el mensaje de disparo, localmente ejecuta la acción, aunque el servidor aun no la haya validado para evitar una sensación de lentitud en el manejo del personaje.

Movimiento

La manera de gestionar el movimiento ha sido modificada varias veces debido a diversas mejoras que se han ido proponiendo a lo largo del proyecto, a continuación, se detallara cada una de ellas y por qué se llevó a cabo, a parte del estado final de la implementación de la sincronización del movimiento entre jugadores.

Inicialmente cada jugador enviaba cada frame la posición, se moviera o no, mientras que localmente la iba actualizando también para evitar la sensación de lentitud de respuesta mencionada anteriormente. El servidor al recibir la respuesta, sabiendo a la velocidad a la que se mueven los jugadores emplea el mismo método mencionado en el disparo, calcula el tiempo trascurrido desde el último paquete y determina donde podría estar el jugador, añadiendo un margen para evitar posibles errores. Si pasa la comprobación, se reenviará el paquete a todos los jugadores.

Debido a la enorme cantidad de paquetes enviados, en muchos casos sin ser necesarios debido a que el jugador no se había movido, se decidió reducir el número de envíos. En el estado final se envía la posición cada 50 milisegundos, es decir 20 veces por segundo, además antes de enviarlo se comprueba que haya cambiado, si no lo ha hecho se enviara un paquete con mucho menor peso indicando que no ha variado. Se eligió esta frecuencia de envío debido a que como se puede observar en la sección de pruebas, al En la versión inicial, se enviaban, dependiendo del hardware empleado, entre 60 y 144 paquetes por segundo, por lo que es una mejora sustancial.

Después de esta implementación cada jugador podía moverse y se validaba su posición correctamente, pero, aunque cada uno veía su movimiento correctamente no ocurría lo mismo con el de los demás.

Por una parte, solo se enviaban 20 veces la posición cada segundo mientras que el juego se ejecutaba de manera estándar a 60 frames por segundo, por lo que mientras que nuestro jugador se movía de manera uniforme, los se movían de forma brusca entre las posiciones recibidas.

Este problema se agravaba si la red no era estable, haciendo que pudiera moverse durante un momento más de lo debido y luego quedarse parado, dependiendo del tiempo de llegada de los paquetes con los datos de la posición.

Interpolación

Para arreglar lo mencionado anteriormente, se desarrolló un sistema de interpolación de posiciones, en el cual, al recibir una posición en vez de mover al jugador permitente directamente a esa posición, se mueve a posiciones intermedias entre el origen y el destino, actualizándola en cada frame de manera que sea cada vez más cercana a la posición destino, de esta manera se consigue el efecto de estar moviéndose de manera fluida. Cada vez que se recibe una nueva posición, se establece como nuevo destino sobre el que ir interpolando la posición entre la actual y la nueva recibida.

El funcionamiento del sistema de interpolación consiste en dividir el número de envíos de posición por segundo entre el número de frames por segundo, para conseguir así un movimiento uniforme independiente de los frames que consiga generar el hardware del usuario.

En el caso probado, el juego funcionaba a 60 frames por segundo, mientras que se enviaban 20 paquetes por segundo, lo que resultaba en que, mediante el empleo de este sistema, la posición se actualiza tres veces más a menudo proporcionando una mejor experiencia al usuario.

Como se puede apreciar en la ilustración 14, si la nueva posición es recibida al final del frame 0, sin interpolación el usuario se moverá inmediatamente a la posición de destino haciendo que parezca que va saltando entre posiciones en vez de moverse, mientras que, al emplear la técnica descrita de la interpolación, el usuario de moverá de forma progresiva mientras espera recibir un nuevo paquete. Como se puede observar el estado final es el mismo en ambos, momento en el cual se debería obtener la siguiente posición destino, por lo que se volvería a repetir el mismo funcionamiento. [16]

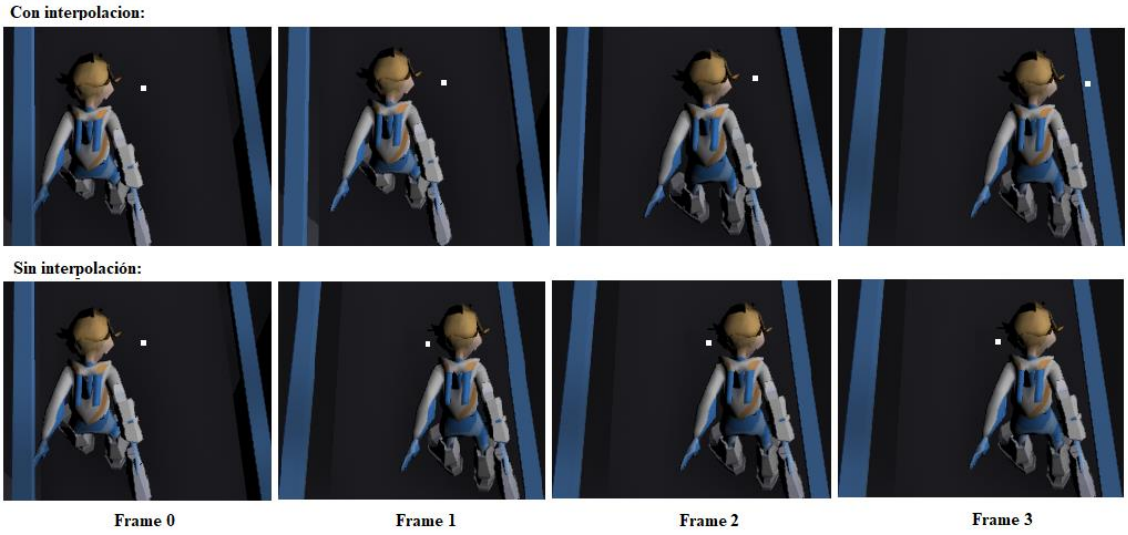

Ilustración 14 Ejemplo del funcionamiento de la interpolación

Extrapolación

A parte del problema con el movimiento de los demás jugadores ya solucionado, se encontró otra dificultad debido a los problemas inherentes a la red.

Si algún paquete llegaba tarde o se perdía, ni con el desarrollo inicial ni con la mejora desarrollada para el movimiento, se podría hacer algo para evitar ver al jugador parado hasta que se reciba el siguiente paquete con la posición.

Si un paquete llega antes de lo debido, no hay ningún problema, simplemente la posición será actualizada antes de lo esperado, debido a que la posición se interpola cada frame no se notará un gran salto. En cambio, si el paquete llega tarde, el tiempo de espera hasta recibirlo provocara que el jugador al que pertenece se quede parado esperando la nueva posición.

Para intentar paliar este efecto, se ha implementado un sistema conocido como extrapolación.

Consiste en, una vez hemos llegado a la posición destino, deberíamos haber recibido la siguiente posición para seguir con la interpolación como se ha descrito anteriormente. De no ser así, se realizará una extrapolación intentando predecir el movimiento que va a realizar los demás usuarios.

Si un jugador se está moviendo en una determinada dirección y hay un problema con algún paquete determinado, es mucho más probable que se usuario haya seguido la misma dirección a que haya variado. [17]

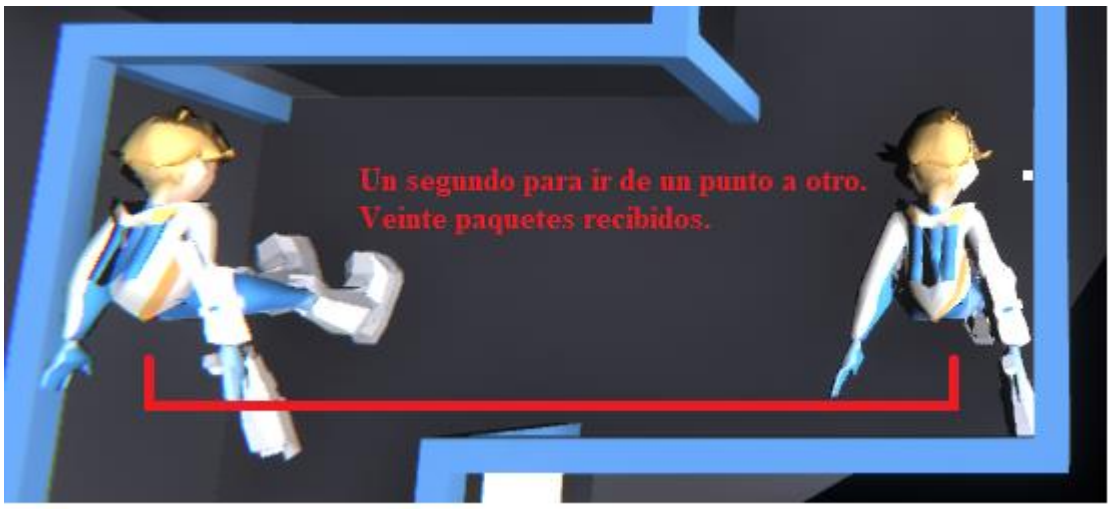

Ilustración 15 Paquetes y movimiento

En la ilustración 15 se puede observar el espacio recorrido en un segundo, para una distancia relativamente pequeña se han necesitado 20 paquetes de movimiento en los que no variaba la dirección. Debido a que los usuarios deben avanzar en el nivel, habrá muchos paquetes de manera seguida sin variar la dirección para poder desplazarse. Si de media un usuario cambiara de dirección cada segundo, la probabilidad de que se fallara la predicción seria de 1 entre 20, por lo que es bastante probable acertar.

Por lo tanto, si no se recibe el paquete cuando el usuario ha alcanzado la posición objetivo, se dará por hecho que sigue moviéndose en la misma dirección por lo que seguiremos moviéndolo en la misma dirección y con la misma velocidad durante cada frame.

Finalmente, al recibir el paquete esperado si se ha acertado con la predicción, el jugador no notará nada debido a que se volverá a aplicar el sistema de interpolación desde la posición actual predicha hasta la posición destino, que serán similares al haber predicho bien.

En caso de fallar en la predicción, al recibir el paquete volverá a actuar el sistema de interpolación, pero esta vez al estar más lejos de la posición deseada y tener que llegar a ella antes de recibir el siguiente paquete, el jugador se moverá más rápido para intentar situarlo en el lugar en la que realmente debería estar.

De manera general, aun fallando la predicción si solo se ha retrasado o perdido un paquete, será imperceptible para el usurario debido a los dos sistemas de interpolación y extrapolación implementados. El único caso notable será cuando se pierdan varios paquetes seguidos y además varié el movimiento, notando así el usuario un ligero aumento de movimiento del jugador pertinente para situarse en la posición en la que debería estar.

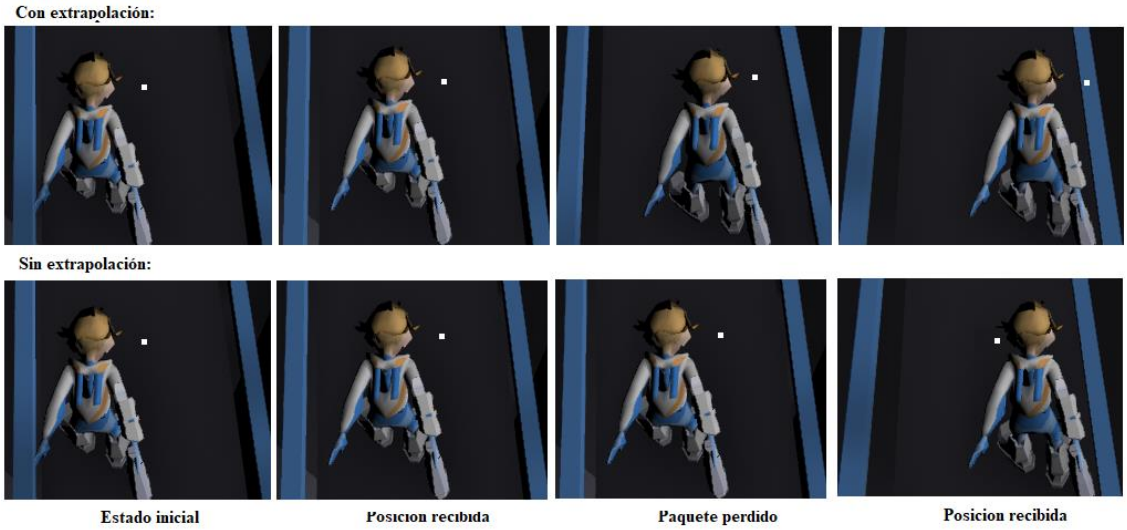

Ilustración 16 Ejemplo de xtrapolación.

En la ilustración 16 se puede ver como al perderse un paquete, mediante la extrapolación no se ha sufrido ningún percance en el movimiento, en la imagen tres el jugador sigue moviéndose a pesar de no haber recibido ningún paquete que lo indique mientras que sin este mecanismo se quedaría quieto. Finalmente se recibe una nueva posición que confirma el acierto en la precisión por lo que no se notara cambio alguno si se ha extrapolado, al contrario que sin ella, notándose un gran salto.

Mediante los métodos explicados anteriormente, se ha conseguido lograr un sistema que no emplea una elevada cantidad de bytes para transmitir los datos necesarios y a su vez simula un comportamiento en el cual parece que no hay ningún retraso entre los demás jugadores y que la posición de cada jugador se actualiza más veces que la cantidad de posiciones recibidas, logrando una experiencia de juego satisfactoria. Incluso con una latencia muy elevada, el videojuego se verá fluido, solo que tardaran más tiempo en verse reflejadas las acciones de cada jugador.

Para intentar paliar el problema del problema derivado de la pérdida o retraso de paquetes, se intentó otra solución que consistía en ir almacenando las posiciones y no mover al jugador hasta tener dos posiciones futuras guardadas, por lo que si no recibíamos una nueva, se podía ir moviendo a la segunda que se había guardado, este

sistema ofreció peores resultados que la extrapolación de movimiento por lo que se desechó, se profundizara sobre las razones en el apartado 8.3 Pruebas de carga de datos.

Salto

Debido a que el salto es un tipo de movimiento especial que requiere una animación al ejecutarse que no se puede cancelar, para evitar problemas al moverse y saltar a la vez, pudiendo solaparse las animaciones, se decidió emplear un paquete para determinar cuándo un jugador salta.

Se realizo una implementación muy similar a la del movimiento. En este caso no hay que interpolar ninguna posición, cuando el jugador salta, aparte de enviarse el paquete correspondiente se ejecuta localmente la animación de salto cancelando todas las demás animaciones activas. Cuando otro cliente recibe ese paquete, simplemente ejecuta la animación de salto por lo que no hace falta ningún sistema para mejorar la fluidez del mismo.

Además, al tener un paquete propio, se evitan problemas de confundir un salto con una subida rápida en vertical por una rampa, o una plataforma elevadora que al no haber saltado no debería haber animación.

6.4 Implementación de los enemigos

Después de bajar diferentes opciones para implementar un sistema de IA para el movimiento de los enemigos, se decidió emplear el sistema de navegación y búsqueda de caminos de Unity.

Primero, se declara sobre el escenario que zonas son transitables por los enemigos. Debido a que el escenario se compone de gran variedad de elementos, se tuvieron que ajustar manualmente muchos de ellos porque, aunque se puede crear la zona automáticamente, muchos caminos estrechos que deberían ser transitables no lo eran, igual que muchas plataformas al estar separadas del mapa principal no se añadían correctamente a toda la zona.

Finalmente se decidió crear de manera manual toda la zona perteneciente a los enemigos, aunque incrementara sustancialmente el trabajo, para asegurarse de que no hubiera ningún problema en el futuro. Asegurándose de que las puertas y los caminos son todos lo suficientemente grandes como para que todos los enemigos logren pasar.

A parte se creó otra zona a parte que contenía todas las plataformas separadas para los enemigos que puedan saltar, haciendo especial hincapié en la distancia entre estas puesto que, si es demasiado grande, los enemigos se quedarían sin poder pasar entre ellas. En la ilustración 15 se puede apreciar el área navegable y las diferentes alturas.

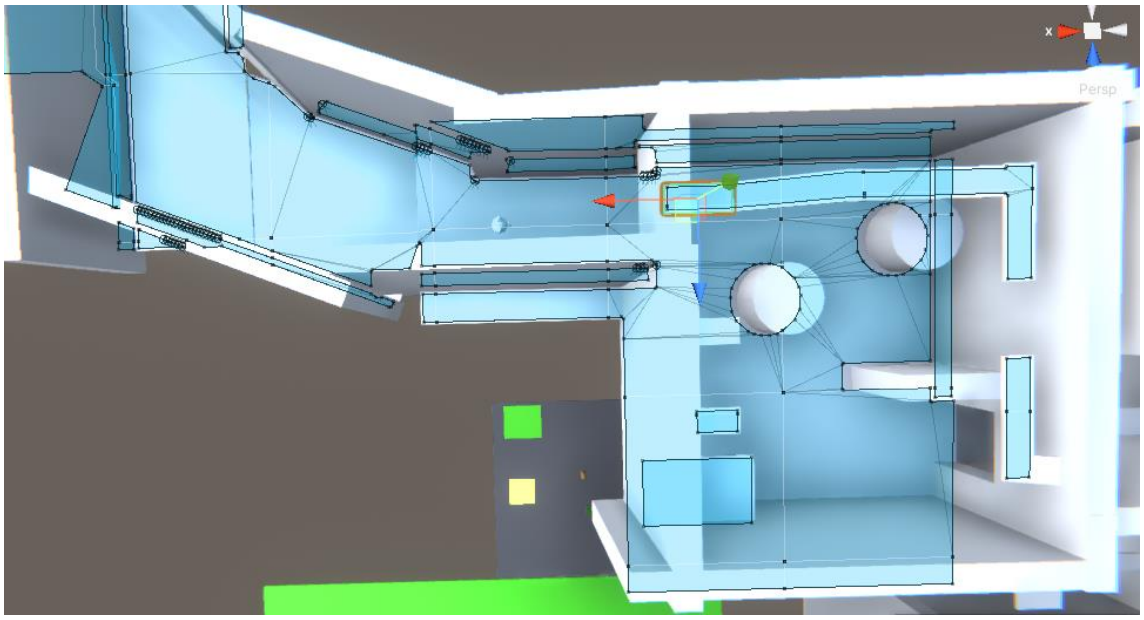

Ilustración 17 Mapa de las zonas transitables por los enemigos marcadas en azul.

Una vez ya se había definido la zona, se comenzó a crear los agentes inteligentes que iban a controlar a los enemigos. [18]

En el proyecto se distinguen dos tipos de enemigos, el tanque y el gato.

6.4.1 **Tanque**

Este enemigo es el de mayor tamaño, debido a esto hay pasillos más estrechos por los que no podrá moverse. También se mueve de forma más lenta y no puede subir a ciertas plataformas por su elevado peso. Finalmente, aunque su disparo hace el doble de daño a los jugadores y la bala disparada es más grande, su cadencia de es mucho más elevada porque lo tarda más en disparar.

Para simular el comportamiento descrito se creó un agente en Unity, con un tamaño superior y con la distancia de salto a cero.

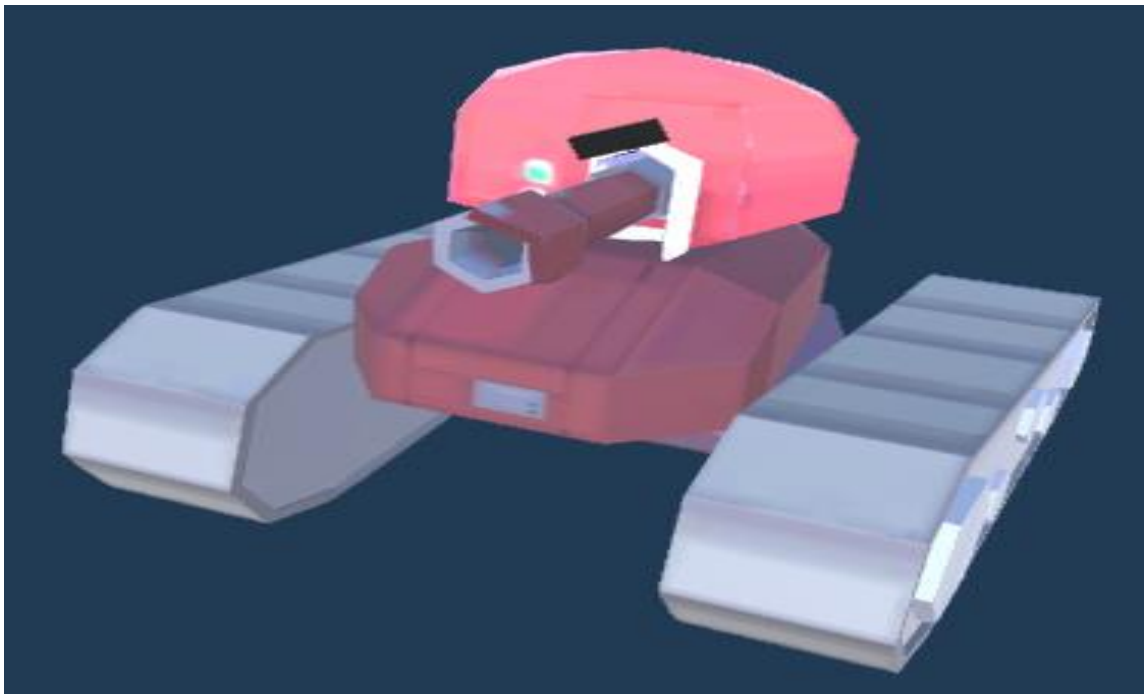

Ilustración 18 Enemigo tanque.

6.4.2 **Gato**

Este enemigo es totalmente opuesto al tanque, tiene un menor tamaño por lo que puede navegar por todo el mapa, a su vez se mueve más rápido. Dispara más rápido que el tanque, pero sus disparos hacen menos daño y son más pequeños y fáciles de esquivar. Debido a la agilidad de este enemigo te seguirá allá donde vayas, incluso saltando sobre las plataformas.

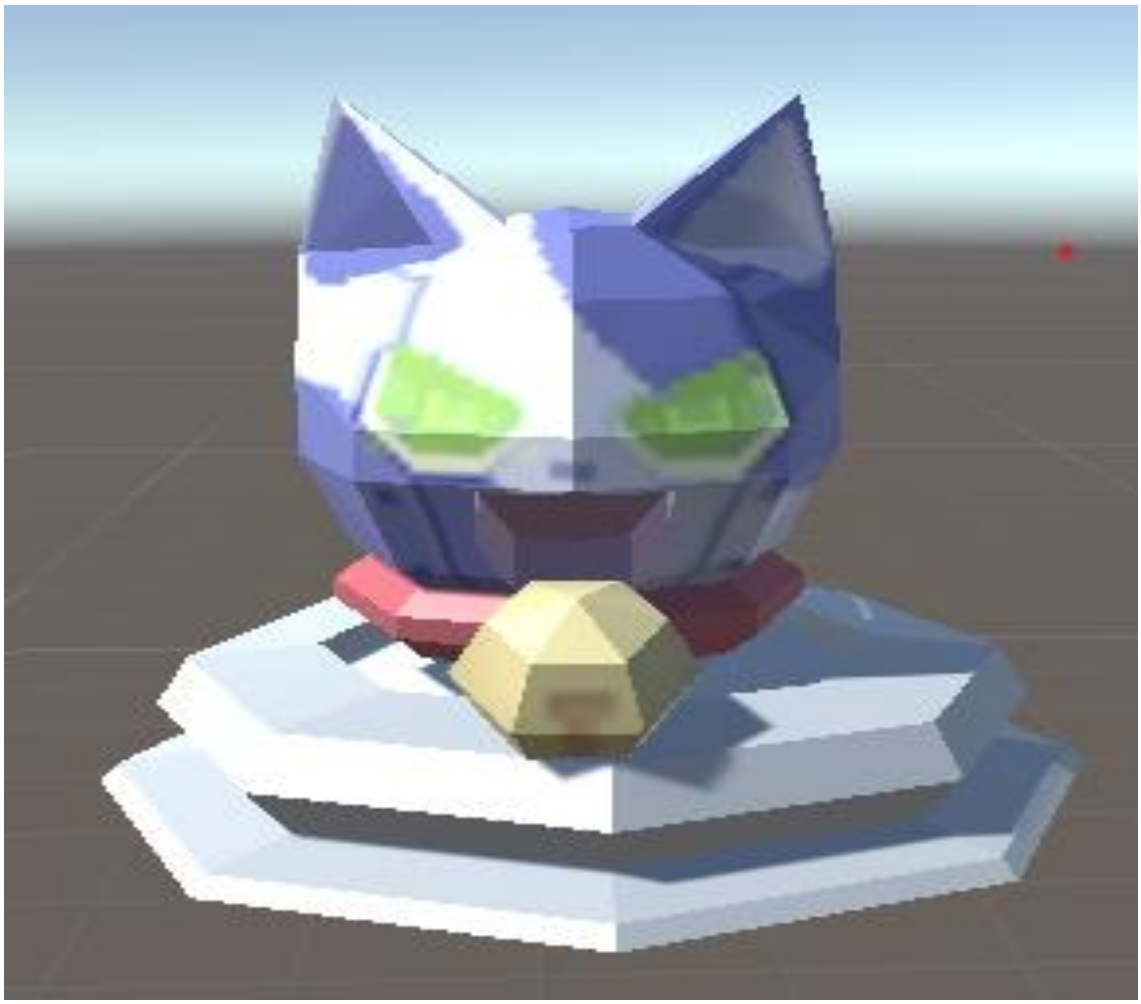

Ilustración 19 Enemigo gato.

6.4.3 **Comportamiento de los enemigos**

Mediante un algoritmo A* de búsqueda de caminos, los enemigos iban hacia el jugador esquivando los obstáculos. Se declaró una distancia máxima del jugador, a partir de la cual estos empezaban a moverse hasta reducir esa distancia por debajo del valor definido.

Una vez están a rango del jugador se comprueba si hay algún objeto entre ellos y el jugador, en caso de ser así se sigue acercando empleando A*. Esto se ha definido así para evitar que se queden parados disparando a una pared si el jugador está detrás.

Si no hay nada entre el jugador y el enemigo, este último comenzará a disparar hasta que el jugador se mueva, lo que provocará que este lo siga, o hasta que el jugador le dispare y elimine al enemigo.

Como se puede ver en el apartado de pruebas, ambos recorrían correctamente el espacio creado.

6.5 Añadido de funcionalidades para los jugadores

Los jugadores quieren empezar una partida lo más simple y rápido posible. Por eso se desarrolló un sistema conocido como matchmaking, que encarga de emparejar a los jugadores que desean jugar y juntarlos en la misma partida.

En la fase inicial de desarrollo, cada jugador creaba una partida y otro jugador debería elegir unirse. Este sistema funciona bien si el usuario quiere elegir a que partida unirse, pero la mayoría de ellos prefieren darle a un botón y que el sistema automáticamente seleccione una partida. Debido a esto se rediseño el sistema para desarrollar una funcionalidad que se encargue de comparar la partida que busca cada usuario y una vez haya dos que hayan elegido el mismo tipo de partida, juntarlos y redireccionarlos a un servidor de juego.

Para implementar esto, cada jugador cuando pulsa el botón jugar, enviara también un paquete que contiene el tipo de partida que desea jugar. Este contendrá una lista con cada una de las características que haya elegido el jugador, ya sea el número de jugadores de la partida o la dificultad.

Cuando un nuevo usuario busque partida, se comparará la lista de características que ha elegido con las de los demás usuarios que están esperando, si coincide con alguno se les reenviara a un servidor de juego donde comenzaran la partida.

Por ejemplo, hay un jugador en cola que ha seleccionado dos jugadores y modo difícil, al rato se añade a la otra otro jugador con modo dos jugadores y fácil. Como estos jugadores han solicitado características diferentes, no se creara ninguna partida y seguirán esperando. Finalmente se une otro con solamente el modo dos jugadores, debido a que sus requisitos coinciden totalmente con algún jugador de los que está en la cola, se le juntara con el que lleve más tiempo esperando. A su vez dado que uno de ellos quiere el modo fácil, la partida se pondrá en fácil porque el otro no ha querido elegir dificultad.

Si lo que se quiere es jugar lo más rápido posible, el jugador no debería seleccionar ninguna opción y simplemente darle a jugar.

A su vez, el diseño está pensado para añadir en un futuro cualquier parámetro adicional, como la latencia al algoritmo de matchmaking. Haciendo así que los jugadores tengan una mejor experiencia de juego porque solo se les emparejara si tienen latencias no muy dispares.

A parte, como ya se ha comentado, si un jugador desea buscar partida deberá pulsar el botón de jugar. Si estando se vuelve a pulsar el botón, el sistema comprobará si se ha cambiado alguna característica en la partida buscada, como la dificultad. De ser así se actualizará la lista de requisitos de la partida de ese jugador sin quitarlo de la cola. En cambio, si no ha variado nada, el jugador será quitado de la cola. Si un cliente estaba en

cola y se desconecta, será también eliminado de esta para evitar emparejar a un jugador que ya no está. [19]

7. Pruebas

Debido a que se ha implementado una arquitectura con diversa funcionalidad, es necesario hacer varias pruebas pues es probable que, aunque localmente se vea correctamente, debido a la naturaleza de la red puede haber errores sin detectar.

7.1 Pruebas de movimiento

Primero se probó a enviar 60 paquetes por segundo y sin implementar ningún sistema de mejora del movimiento.

A pesar de enviar tantos datos, el movimiento no era fluido. El sistema donde se probó no era capaz de gestionar exactamente un paquete por segundo por lo que el usuario apreciaba saltos en el movimiento.

Debido a esto, como se detalla en la sección 7.3 Sincronización de los estados de los jugadores, se implementó la interpolación y la extrapolación.

Al volver a probar con estas mejoras, el movimiento se veía correctamente por lo que se llevaron a cabo pruebas más exhaustivas.

Mediante una opción de la API para retrasar la llegada de paquetes, se probó a aumentar la latencia. Hasta un máximo de 200 milisegundos no se aprecia ningún cambio para el jugador, si se aumentaba más empieza a dar la impresión de que los demás jugadores se mueven con retraso. Este dato se puede usar para en un futuro limitar el ping de los jugadores a 200.

También se realizaron test de perdida de paquetes mediante otra opción en la API que lo permite. Hasta un 20% de paquetes perdidos, no se puede apreciar ningún cambio. A partir de este umbral se empiezan a apreciar ciertos saltos entre posiciones. Viendo en este caso en acción el sistema de corrección de posición al fallar la extrapolación, mientras que la mayor parte del tiempo no se llegaba a apreciar un cambio, cuando coincidía la perdida de paquetes con el cambio de dirección, se podía ver como el jugador se desplazaba rápidamente para ir a la posición en la que está realmente.

Debido a que en un entorno real la latencia no suele ser tan elevada y no se pierden tantos paquetes, se considera que es apto para llevar a cabo pruebas con los clientes finales.

7.2 Pruebas de clientes maliciosos

Se realizaron pruebas para activar elementos del escenario sin estar cerca de ellos. El servidor al comprobar que la distancia del jugador era demasiado grande con ese objeto, no retransmitía ese paquete por lo que ningún jugador veía esa acción.

También se realizaron pruebas intentando mover de forma rápida al jugador, igual que en el párrafo anterior el servidor consideraba que el desplazamiento era demasiado elevado, por lo que este paquete tampoco era retransmitido.

Se realizó las mismas pruebas con el disparo, intentando que la posición desde la que se dispare la bala sea muy diferente a la del jugador, si esta lo era, ocurre lo mismo que en los casos anteriores.

Finalmente, con el saltó simplemente se probó que una vez un jugador hubiera saltado no podía volver a saltar.

Después de probar con un cliente normal y uno malicioso, se estableció un intervalo de un 20% en la distancia. Durante el desarrollo de una partida normal nunca se obtuvo un dato que diferiría en más de un 20% sobre el que tuviera que ser. Por ejemplo, la diferencia posición en la que el jugador dice está y en la que puede estar nunca era más grande que un 20%, siendo de media un 5% debido al tiempo en procesar el paquete y a diferencias propias de la implementación de los números en coma flotante, al no ser una representación exacta se pueden acumular errores de posición que son corregidos al recibir el nuevo paquete.

7.3 Pruebas de cantidad de datos enviados

Inicialmente, se enviaban 60 paquetes de posición por segundo. Los demás paquetes son despreciables para contar la cantidad de datos enviados, puesto que contienen mucha menos información y no son enviados tan a menudo.

Debido a la creación de todos los sistemas relativos al movimiento, es posible reducir este número sin que cause problemas.

Se probo a ir reduciendo hasta que el usuario notara algún cambio. La opinión general de los integrantes del equipo fue que, al ser menor de 20, podían aparecer pequeños saltos en la posición que se acentuaban cuanto menor era este número, por lo que se dejaron en 20 envíos por segundo. Reduciendo así en dos tercios la cantidad de información enviada.

Este paquete contiene 4 bytes de overhead, 4 bytes por coordenada de la posición, siendo en total 12 por ser en 3d el personaje, otros 16 bytes pertenecientes a la rotación

y finalmente 8 bytes para indicar la dirección en la que desea moverse el jugador, siendo en este caso solo necesarias dos coordenadas porque no puede moverse hacia arriba. Esto da un total de 40 bytes por paquete enviado.

Por lo tanto, se enviarán 40*20 bytes por segundo, lo que viene siendo 0.8 Kbps. A esta cantidad hay que sumarle datos propios del protocolo UDP, del tipo de canal empleado para enviar los paquetes y todo lo demás enviado, incluyendo el paquete keepalive.

Las pruebas de carga de red daban una media de 3 Kbps enviados con dos jugadores, lo que no es una cantidad muy elevada y puede ser gestionada por cualquier cliente.

Al haber dos jugadores, la cantidad de datos enviada y recibida es similar, pero cuando crece el número de jugadores, los datos recibidos también lo hacen de la misma forma por lo que, aunque este número sigue siendo perfectamente asumible con varios jugadores, es un dato a tener en cuenta si se quieren crear partidas de cincuenta o cien jugadores simultáneos, cosa que no ocurre de momento en el proyecto.

7.4 Pruebas de carga del servidor

Para asegurar que el servidor era capaz de alojar a un número elevado de jugadores, primero se conectaron cien clientes con el servidor maestro. Esto fue realizado suprimiendo el patrón singleton que impedía que un proceso tuviera más de un cliente y se lanzaron cuatro procesos, cada uno con veinticinco clientes.

El servidor maestro apenas creció en uso, debido a que su función es simplemente interconectar a los usuarios, por lo que no necesita una gran capacidad de cálculo, situándose en un 2% sin ningún cliente conectado y en 4% con cien clientes activos, por lo que se perfectamente capaz de mantenerlos.

Después se hizo lo mismo con el servidor de juego, se crearon veinte partidas simultaneas con dos jugadores cada una. Como actualmente solo se dispone de un servidor, todas se crearon el mismo. Cada una de ellas tenía un uso medio entre 2% y 5% por lo que, aunque podía manejarlas correctamente, se limitó el número de partidas máximas a veinte para evitar problemas puntuales.

Finalmente, se comprobó la cantidad de datos transmitida. Siendo en el servidor maestro muy baja puesto que la mayoría del tiempo no ocurre nada. Mientras que en el servidor de juego se mantenía en aproximadamente 10 Kbps con picos de 15 Kbps, debido a que solo había dos jugadores simultáneos.

7.5 Pruebas de flujos

Debido a la que una de las partes importantes era que el sistema fuera tolerante a fallos, se incluyeron diversas opciones para reconectar al jugador en caso de caída o cerrar el servidor de juego y redirigir a los clientes al servidor maestro en caso de error crítico.

Se ha probado a reiniciar al servidor mientras un cliente está conectado, aunque este se desconecte, puesto que intentara volver a conectarse otra vez, logra recuperar la conexión por lo que podrá seguir jugando sin percances.

También se probó a desconectar un cliente de la red temporalmente. Si el tiempo es menor de diez segundos no ocurre nada puesto que la conexión no se cierra, en cambio si supera este valor el cliente cerrará la conexión y volverá a conectarse exitosamente.

Finalmente se generó un error irrecuperable en el servidor de juego lo que provocó que este se cerrara de forma brusca. En este caso al pasar diez segundos sin respuesta del mismo, los clientes volvieron al servidor maestro para poder jugar otra partida. A su vez se generó un registro con el problema que ha provocado el cierre del servidor para su posterior revisión.

7.6 Pruebas de los enemigos

Es importante que los enemigos se muevan de manera correcta, de no ser así se podrían provocar errores u ofrecer una mala experiencia al usuario al ver cuando un enemigo realiza acciones que no se espera.

Primero se colocaron objetos en la trayectoria de un enemigo y se le puso un destino para que fuera a él. Como se puede ver en la ilustración 20, se consiguió que llegara al destino evitando los obstáculos.

Se repetido esta prueba con diferentes escenarios y enemigos. En todos ellos se logró el resultado esperado, alcanzando el objetivo correctamente.

Finalmente, debido a que los enemigos tienen que acercarse a una distancia determinada y luego disparar siempre que el jugador permanezca en rango. Al probarlo, una vez estaba lo suficiente mente cerca del jugador, el enemigo disparaba. Si el jugador se alejaba fuera del rango el enemigo le seguía, por lo que funciona como se esperaba.

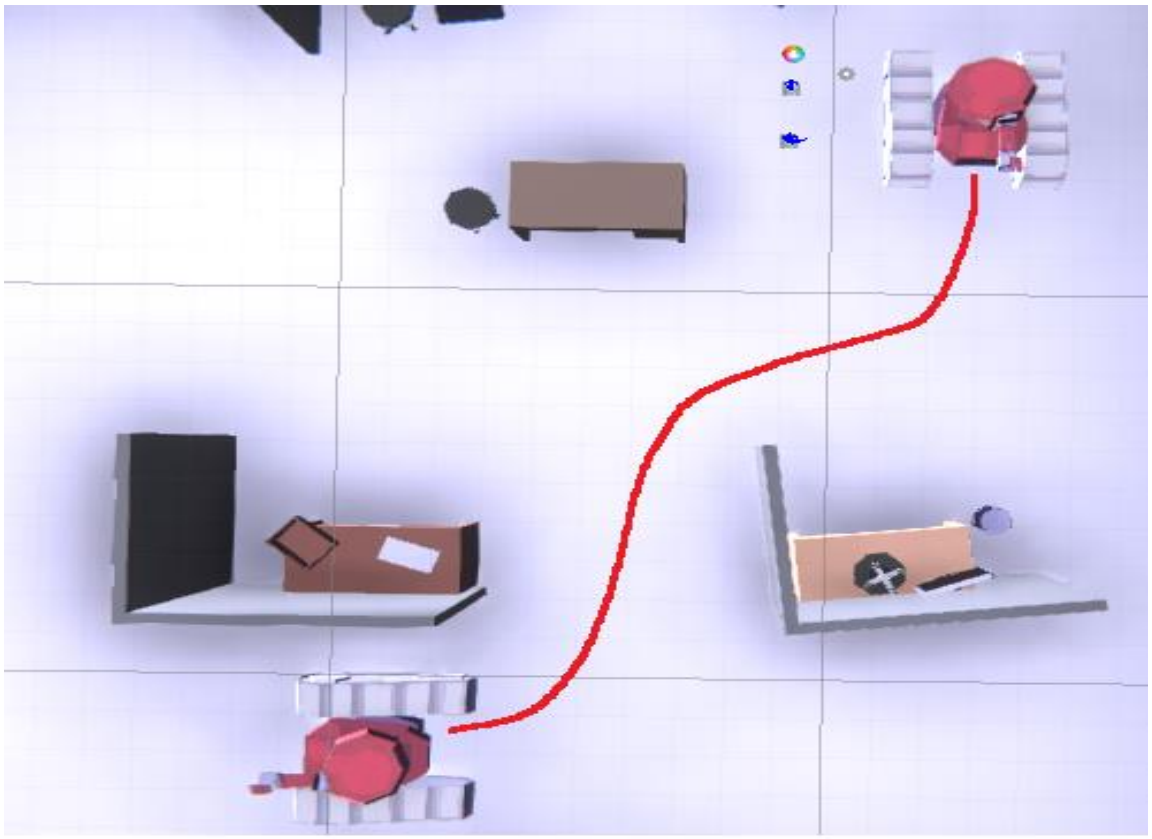

Ilustración 20 Sistema de búsqueda de caminos.

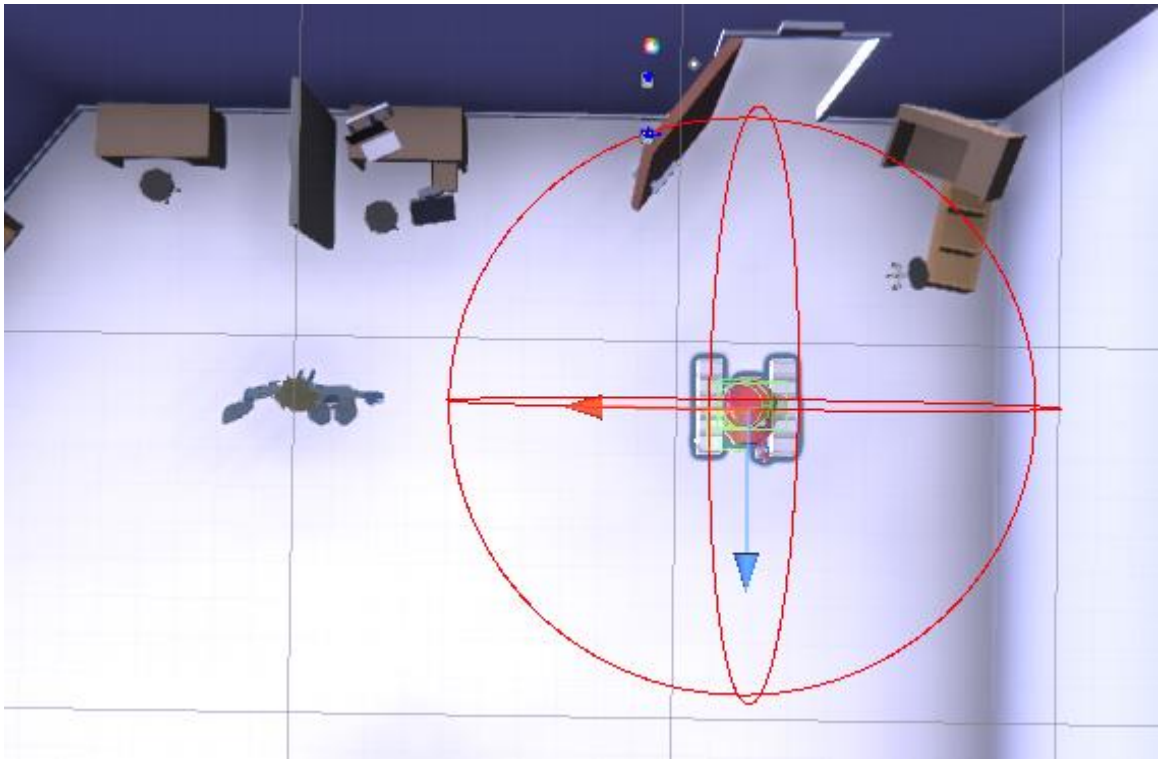

Ilustración 21 Rango de disparo.

8. Conclusiones

Se han conseguido completar la mayoría de los objetivos propuestos.

- Forzar la cooperación entre los jugadores para superar el juego.
- Permitir que los usuarios jueguen entre sí.
- Implementación de sistemas para lograr un movimiento fluido.
- Desarrollo de una arquitectura escalable con tolerancia a fallos de desconexión y perdida de paquetes.
- Protección contra clientes maliciosos.
- Sistema de emparejamiento de jugadores automático
- Creación de una inteligencia artificial que permita a los enemigos moverse por el escenario evitando los obstáculos y siguiendo al jugador.

En cambio, hay un objetivo que no se ha cumplido.

- Creación de un jefe final con diversas mecánicas que hagan hincapié en la cooperación.

En general la mayoría de los objetivos han sido completados satisfactoriamente. A pesar de que haya un objetivo no logrado, no se considera prioritario, por lo que como se tiene un videojuego completamente funcional, se considera un éxito.

Se han superado muchas dificultades, desarrollos iniciales del movimiento que parecían simples, han resultado ser enormemente complejos, pero finalmente se ha dado con una solución aceptable.

A su vez se han generado muchos problemas al intentar sincronizar estados sobre un entorno imperfecto como la red, teniendo que usar mecanismos de ordenación y reenvió de paquetes para asegurar la llegada o el orden de estos en mensajes cruciales.

Después del desarrollo, se ha aprendido a estimar mejor la duración de las tareas, puesto que muchas de ellas han duplicado o triplicado su estimación inicial. Ahora se tiene mejor criterio a la hora de evaluarlas en un futuro.

Además, se ha aprendido a trabajar en grupo, teniendo que compartir ideas entre diversos miembros del equipo intentar llegar a un punto común.

Finalmente, ahora se conoce la dificultad del desarrollo de sistemas en red, complicando enormemente cosas que de forma local son tribales. Ya no solo por la sincronización de estados, sino por la enorme cantidad de recursos destinados solo a parar clientes maliciosos que puedan entorpecer la experiencia de los demás.

Finalmente, se han aprendido los sistemas usados por los videojuegos multijugador para solucionar los múltiples problemas generados, desde algunos sencillos como validar las acciones de los usuarios, hasta otros mucho más complejos como retroceder

9. Trabajo Futuro

Durante el desarrollo del proyecto han ido surgiendo ideas, que por no ser prioritarias se han aplazado, pese a ello toda la arquitectura desarrollada ha sido pensando en los futuros añadidos por que se pueden implementar de forma sencilla.

Se han pensado las siguientes.

Sistema de chat

Debido a que el juego se basa en la cooperación, sería interesante añadir un método de comunicación escrita para que ambos jugadores se entendieran mejor. También podría ser interesante añadir un chat por voz para ver si mejora la coordinación entre los mismos.

Sistema de amigos

El sistema para emparejar jugador actual es automático, si dos amigos desean juntos la única opción que tienen es intentar meter cola a la vez para que les una a la misma partida. Un buen añadido sería crear un sistema de amigos para que los usuarios puedan jugar con quien deseen.

Persistencia

En el estado actual, los jugadores se conectan de manera anónima al servidor por lo que una vez desconectados, lo que habían obtenido se pierde. Se podría implementar un sistema de registro junto a una base de datos que permitiera tener a cada usuario una cuenta logrando así la impresión progreso puesto que todo lo realizado se guarda cada vez.

Sistema de puntuación y clasificación

Una manera de hacer regulable un juego es añadir un sistema de puntuación que permita a los usuarios ver los puntos obtenidos en cada partida e ir comparándolos con los demás. Logrando así una competición con posibles premios a los ganadores.

Una vez implementada la puntuación, se podría usar esta para emparejar a los usuarios entre sí, logrando partidas más igualadas

Más variedad de enemigos y un jefe final

Para que cada partida parezca diferente, se ha pensado implementar en un futuro diversos enemigos que parezcan de manera aleatoria, logrando así otorgar una mejor experiencia de juego al no ser tan repetitivo para el usuario.

10. Bibliografía

[1] https://smartlaunch.com/q2-2021s-most-impactful-pc-games/ [19/07/2021]

[2] https://venturebeat.com/2021/07/04/newzoo-game-market-will-hit-200b-in-2024/ [20/07/2021]

[3] https://es.wikipedia.org/wiki/Portal_2 [10/04/2021]

[4] https://es.wikipedia.org/wiki/League_of_Legends [10/04/2021]

[5] https://leaguefeed.net/did-you-know-total-league-of-legends-player-count-updated [12/04/2021]

[6] https://gafferongames.com/post/networked_physics_in_virtual_reality/ [20/04/2021]

[7] https://blog.unity.com/technology/choosing-the-right-netcode-for-your-game [01/05/2021]

[8] https://forum.unity.com/threads/mirror-or-mlapi.1082363/ [02/06/2021]

[9] https://docs.unity3d.com/es/530/Manual/UNetUsingHLAPI.html [03/06/2021]

[10] https://forum.unity.com/threads/what-are-the-pros-and-cons-of-available-network-solutions-assets.609088/ [03/06/2021]

[11] https://github.com/RevenantX/LiteNetLib [05/06/2021]

[12] https://softwareengineering.stackexchange.com/questions/342254/tcp-or-udp-for-a-multiplayer-game [05/06/2021]

[13] https://docs.unity3d.com/Manual/UNetClientServer.html [06/06/2021]

[14] https://gafferongames.com/post/reliable_ordered_messages/ [08/06/2021]

[15] https://www.gabrielgambetta.com/client-server-game-architecture.html [08/06/2021]

[16] https://answers.unity.com/questions/767134/physics-what-is-interpolate-extrapolate-discrete-c.html [12/06/2021]

[17] http://www.theappguruz.com/blog/extrapolate-position-in-unity [12/06/2021]

[18] https://learn.unity.com/tutorial/unity-navmesh#5c7f8528edbc2a002053b499 [20/06/2021]

[19] https://medium.com/@thasorn52111/game-design-102-matchmaking-c109adac623a [25/06/2021]

11. Glosario

Matchmaking

Sistema presente en la mayoría de videojuegos multijugador que trata de enfrentar a jugadores o equipos de jugadores de un nivel o habilidad similar, analizando sus estadísticas de anteriores partidas para establecer el rango de cada uno.

Servidor autoritativo

Tipo de servidor en el que la partida se desarrolla en el mismo, los clientes serán espectadores que podrán interactuar con la partida de manera limitada. Debido a la restricción de interacciones y a la unificación de los datos en un entorno seguro como es el servidor, se consigue evitar cualquier manipulación, puesto que, si algún cliente altera la partida, será solo su representación y no la real.

Latencia

Tiempo que tarda en transmitirse un paquete dentro de la red, ya sea por costes de transmisión, procesado de paquetes o de cualquier otro tipo.

API

Es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones.

Api de alto nivel

Api que emplea métodos más cercanos al hardware. Se encarga del nivel más bajo de la capa de transporte, asegurándose de enviar y recibir paquetes correctamente. No tiene ninguna funcionalidad referente al multijugador

Api de bajo nivel

Api construida sobre una capa de transporte de menos nivel. Añade funcionalidad sobre esta **y** maneja las tareas comunes que son requeridas para juegos multijugador. A cambio no permite tanta flexibilidad puesto que gran parte está ya predefinida.

Patrón singleton

Es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase e implementar un único punto de acceso.

Keepalive

Es un mensaje enviado por un dispositivo a otro para comprobar si la conexión sigue vigente, o para prevenir que sea desconectado por inactividad.

Overhead

Sobrecoste de recursos que se deben emplear para llevar a cabo la tarea dada.

Frame

Cada una de las imágenes estáticas que forman parte de la sucesión de imágenes de un videojuego **y** que consiguen crear una sensación de movimiento.

Interpolación

Método para obtener valores intermedios de manera lineal entre dos puntos. En el proyecto permite obtener posiciones intermedias entre las que se mueve un jugador.

Extrapolación

Método para obtener valores más allá del intervalo que marquen dos puntos lineales. En el proyecto permite obtener la posición hacia la cual se dirige un jugador.

12. Anexo

Anexo 1 Game design document

Anexo: Game Design Document
Project Biobot

Diseñador y desarrollador	Rodrigo Martinez Machuca, Héctor, Ana
Plataforma	PC
Versión	1.0
Sinopsis	Project Biobot es un juego cooperativo que nos mete en un universo futurista donde nuestros protagonistas son Biobots, personas que han sido generado de manera artificial para cumplir las órdenes de una IA, pero que consiguen escapar de sus garras. En un mundo el cual aún no se sabe lo que está sucediendo y contra que o quien están luchando.
Categoría	Acción, ThirdPerson Shooter, Plataformas
Licencia	
Mecánica	El jugador manejara a uno de los personajes principales y se encontraran con diferentes dificultades, entre enemigos que habrá que vencerles con disparos y con plataformas para superar o puzles.
Tecnología	Unity 5, C#
Publico	Todo el publico
Niveles	Nos encontraremos con un nivel con unas cuantas mecánicas, pero con la idea de implementar más para un futuro y finalizar la historia.

Anexo 2 Diagrama de Gantt de los Sprints

https://drive.google.com/file/d/1T-HILzqIT6baJElvdu5p7lKTtr3A_WUC/view?usp=sharing

Anexo 3 Script para compilar todos los servidores y el cliente

https://drive.google.com/file/d/1pqlpEjUJMKQOUgtf6Q_3eya0_ZTJN6kl/view?usp=sharing

Anexo 4 Código declarando todos los paquetes

https://drive.google.com/file/d/1DI0oMSIeU-M_WiY2U2kyeUBXv8wRm99T/view?usp=sharing

Anexo 5 Código del servidor maestro

https://drive.google.com/file/d/1GXQVtbRk9_vwimbQFjbtCnu3f_tUC3WL/view?usp=sharing

Anexo 6 Código del servidor de juego

https://drive.google.com/file/d/17eJK5etSY05zgv7CsbjkTns7__XH1V2n/view?usp=sharing

Anexo 7 Código del cliente

https://drive.google.com/file/d/1ywCwKWKYvAsZkMDeKxbjg9rQZmx6ezxo/view?usp=sharing

Anexo 8 Video del juego

https://drive.google.com/drive/folders/1M4Ca0rDcXenif2z6jX5KoE6KE2eGkdxR?usp=sharing