Escuela Politécnica Superior de Linares

Grado en Ingeniería Telemática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Linares

Trabajo Fin de Grado

ANÁLISIS DE LAS PRESTACIONES DE UNITY3D PARA EL DESARROLLO DE JUEGOS MULTIJUGADOR

Alumno: **Sergio García Solano**

Tutor: Prof. D. Raquel Viciana Abad
Depto.: Ingeniería de Telecomunicación

Junio, 2019

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Linares

Trabajo Fin de Grado

ANÁLISIS DE LAS PRESTACIONES DE UNITY3D PARA EL DESARROLLO DE JUEGOS MULTIJUADOR

Alumno: **Sergio García Solano**
Firma:

GARCIA SOLANO, SERGIO (FIRMA)
Firmado digitalmente por GARCIA SOLANO, SERGIO (FIRMA)
Fecha: 2019.06.20 10:21:38 +02'00'

Departamento de Ingeniería de Telecomunicación

Tutora: **Raquel Viciana Abad**
Firma:

VICIANA ABAD RAQUEL - 45595778A
Firmado digitalmente por VICIANA ABAD RAQUEL - 45595778A
Fecha: 2019.06.20 10:33:07 +02'00'

Índice General

1. Resumen

En el presente Trabajo Fin de Grado (TFG) se hace uso de un motor de videojuego (*game engine* en inglés) llamado **Unity3D**, el cual proporciona una serie de herramientas para la elaboración de juegos, ya sea este de un solo jugador o para varios jugadores. Uno de los objetivos iniciales es familiarizarse con las herramientas de Unity3D para crear un juego, y posteriormente añadirle la opción de ser multijugador.

Posteriormente, se le incluirá al juego la función de utilizar **WebRTC** para el envío de datos multimedia entre dos usuarios punto a punto. El juego es multiplataforma, ya que funcionará tanto para PC, para dispositivos Android, y contará con la posibilidad de funcionar con las *Oculus Go*.

Finalmente, se realizará un estudio de cómo el juego se comporta en determinados escenarios en donde se va incrementando poco a poco la carga de la red aumentando el número de jugadores, incluyendo jugadores maliciosos y abriendo muchas salas de *streaming*.

2. Introducción

En la actualidad la industria del videojuego aparte de ser una fuente de ingresos en auge desde el punto de vista económico es también una de las industrias que incorporan con mayor rapidez las principales novedades tecnológicas, y, por lo tanto, los últimos desarrollos en el ámbito de las TIC. De hecho, es interesante analizar hasta qué punto las plataformas de desarrollo de videojuegos actuales se adaptan a las exigencias de los usuarios que requieren estar conectados en todo lugar y en cualquier momento. En este sentido, la propuesta de este proyecto es analizar que tecnologías son usadas principalmente para permitir el desarrollo de juegos multijugador que se adapten a los requisitos de intercambio de datos multimedia y en tiempo real desde equipos conectados a redes privadas y mediante el uso de distintos tipos de dispositivos.

El proyecto contará con la posibilidad de que los jugadores puedan comunicarse entre ellos por medio de sus cámaras y micrófonos, así como dotarlos de un chat de texto. Esta tarea ha sido realizada mediante el uso de una API de Unity3D para utilizar el *framework* o marco de referencia WebRTC[1]. Unity3D por sí mismo carece de esta funcionalidad, por lo que ha sido necesario buscar un *asset* gratuito desde la *Asset Store* que proporciona Unity3D. La API que proporciona este *asset* sólo permite la comunicación por medio de mensajes de texto, de modo que uno de los retos del proyecto ha sido adaptar esa API para que también sea capaz de enviar vídeo y audio. Esta tarea ha sido realizada a nivel de aplicación. Se hace uso también de un servidor de señalización proporcionado por la API que ha sido ligeramente adaptado a las necesidades de este proyecto, pero, aun así, la API también tiene un servidor de señalización público, por lo que podemos conectarnos a él desde cualquier lugar con conexión a Internet.

Posteriormente, también se ha adaptado el juego a una versión Android. En la versión Android se ha diseñado dos joysticks para el control del movimiento y la cámara y teclas en la pantalla para que desde el teléfono se pueda también acceder a los servicios de *streaming*. Un jugador que ejecute el juego desde su PC puede jugar contra otro que juegue desde su dispositivo Android.

[1] https://www.w3.org/TR/webrtc/ candidato para evaluación como RFC desde 2018

Finalmente, se hará un estudio del rendimiento de la red en el que se obtiene datos de cómo el juego responde a situaciones atípicas que exceden el funcionamiento normal del mismo. Las distintas pruebas se han llevado a cabo con un número máximo de jugadores coherente y viendo como la adición de cada jugador carga la ejecución interna de la API de Networking. También se ha llevado a cabo la programación de unos jugadores con un comportamiento anormal que sobrecarguen por encima de lo esperado, según la cadencia de interacción de un usuario prototipo, el funcionamiento de la partida, y así estudiar como la API responde a estos sucesos.

Las funcionalidades programadas en Unity3D pueden desarrollarse con tres lenguajes de programación en sus *scripts*: **JavaScript, Boo y C#.** En este proyecto se ha utilizado C# para el desarrollo completo en Unity3D, y un poco de JavaScript para modificar las funciones del servidor de señalización proporcionada por la API de WebRTC para Unity3D.

2.1 Objetivos

En este apartado se describirán los objetivos de este proyecto, los cuáles serán desarrollados con mayor profundidad a lo largo del documento:

- Analizar en profundidad el motor gráfico de Unity3D para el desarrollo de videojuegos y estudiar su API para la incorporación de funciones multijugador o multiusuario en las aplicaciones desarrolladas.
- El desarrollo de un juego completo que haga uso de dicha API, profundizando en aquellos componentes que hayan sido necesarios utilizar para el desarrollo del juego.
- La implementación de un chat de texto con videoconferencia para que los jugadores puedan comunicarse entre ellos cuando lo deseen. Para ello se estudiará el protocolo de tiempo real utilizado, en este caso los usados bajo el *framework* de WebRTC.
- Análisis de las prestaciones desde el punto de vista de vista de la red de la API de *Networking* de Unity3D.

2.2 Estructura del proyecto

Este documento ha sido estructurado en los siguientes capítulos:

1. **Resumen:** Es una página donde se explica a grandes rasgos que se hará en el proyecto.
2. **Introducción:** Se hará una breve descripción del proyecto desarrollado, incluyendo la definición de los objetivos y algunos conceptos básicos.
3. **Descripción del juego:** En este apartado se describirá el funcionamiento básico del juego, se describirá cómo iniciar una partida y qué pantallas se encontrarán en el proceso. Además, se indicará los objetivos del juego para ganar la partida, y qué tipo de funciones puede encontrarse en él.
4. **Fase de desarrollo del videojuego:** Este capítulo se detallará cómo se ha desarrollado el juego para un solo jugador en Unity3D. Se indicarán los componentes utilizados y la programación de los *scripts* que hayan sido necesarios crear.
5. **Funciones de red en Unity3D:** En este capítulo se describirá en profundidad el uso de los elementos de la API de *Networking* que se han utilizado para que el juego sea multijugador. Finalmente, se explica cómo a partir de la API de WebRTC se ha integrado la funcionalidad de transmitir audio y video entre los jugadores activos en el juego.
6. **Interfaces desarrolladas**: En este capítulo se detalla cómo se han desarrollado nuevas interfaces del juego que le permitan la posibilidad de funcionar en dispositivos Android y mediante las Oculus Go.
7. **Pruebas de rendimiento de la red:** Se realizará un estudio del rendimiento de la API de *Networking* mediante la recogida de datos en una serie de escenarios. Además, se comparará el comportamiento de esta API entre una red cableada y otra inalámbrica.
8. **Conclusiones:** Apartado donde se resume los datos obtenidos del proyecto y una breve reflexión sobre Unity3D. Además, se exponen una serie de líneas futuras o mejoras derivadas de este proyecto.

2.3 Antecedentes

Los motores de videojuegos [1], (conocidos como *game engine* en inglés), son una serie de rutinas de programación enfocadas a la creación de un videojuego completo.

Antes de la existencia de los motores de videojuegos, los juegos se desarrollaban desde cero, es decir, no había una separación entre la parte gráfica y la física. De modo, que había que intentar mantener el código lo más simple posible y manipular los objetos, que tras el proceso de renderizado y rasterizado se mostraban píxel a píxel, de manera cuidadosa para evitar un mal rendimiento del juego a causa de las limitaciones que presentaban los *hardware* de aquella época. No fue hasta la época de los 90 hasta que se acuñó el término *game engine.* Juegos como *Doom* o *Quake* acumularon popularidad ya que, en lugar de hacer un videojuego desde cero, se licenciaron los núcleos del mismo para emplearlos como punto de partida y desarrollar sus propios motores de videojuego.

Cuando esta tecnología evolucionó, los motores de videojuego empezaron a abarcar posibilidades, como su uso para el entrenamiento profesional, para la medicina y las simulaciones militares, o la animación de alto realismo en la industria del cine.

La aparición de las API de *DirectX* y OpenGL impulsaron notablemente la tecnología en el ámbito de los videojuegos, ya que son APIs desarrolladas para facilitar las complejas tareas relacionadas con la programación de juegos y vídeo. La primera versión de DirectX fue lanzada en septiembre del año 1995 bajo el nombre de *Windows Games SDK*. A pesar de la creación anterior de *OpenGL* en 1992, *DirectX* tuvo una notoriedad superior en el mundo de los videojuegos y que todavía está presente para Windows 10 en su versión 12.

Generalmente, las compañías han hecho sus propios motores de videojuego. Pero con el paso del tiempo, el coste y la cantidad tan grande de elementos que debían contener estos motores hicieron que varias compañías empezaran a especializarse exclusivamente en construir motores de videojuego y así poder comercializarlos. La razón es porque se requiere mucho tiempo y dinero para que una compañía, además de producir su propio motor de videojuego, lo mantenga con las correspondientes actualizaciones

necesarias, y tenga que desarrollar posteriormente sus propios videojuegos atendiendo a todas las posibles plataformas de ejecución y periféricos disponibles en el mercado.

Los motores de videojuegos son unas de las herramientas actuales más complejas ya que poseen numerosos elementos, y en los últimos años se ha avanzado sobre todo en crear utilidades o *assets* que permitan incorporar la distribución en red de contenidos y la interacción entre multijugadores, y la funcionalidad que les permita ser multiplataforma e integrar y asociar funcionalidades a muy diversos interfaces.

2.4 Estado del arte

En este apartado se hará un análisis de la actualidad de las tecnologías empleadas en el proyecto. Se hablará de los motores de videojuegos más utilizados y las diferentes tecnologías que ofrecen la posibilidad de permitir la conectividad entre jugadores.

2.4.1 Motores de videojuegos en el mercado

El crecimiento en la demanda de videojuegos y la gran mejora que ha experimentado el mundo de la tecnología ha posibilitado la creación de los motores de videojuegos [2]. A continuación, se definen los tres más famosos, aunque haya muchos más:

- **Unity3D:** [3] Es el motor utilizado en este proyecto. Está desarrollado por *Unity3D Technologies* y se puede utilizar en múltiples plataformas de desarrollo como Microsoft Windows, OS X y Linux. Ofrece muchas posibilidades a la hora de decidir hacia qué plataforma está destinada el producto que diseñemos, y trae también la posibilidad de incluir dispositivos de realidad extendida (Oculus GO, HTC Vive, PlayStation VR…entre otros). En el actual proyecto se ha desarrollado el juego tanto para PC como para Android. Además, posee dos versiones, una de pago llamada Unity3D Pro y otra gratuita llamada Unity3D Personal.

- **Unreal Engine:** [4] Es un motor muy potente diseñado para PC y consolas. Fue desarrollado inicialmente para *shooters* en primera persona[2], pero ha sido utilizado también para numerosos géneros a lo largo del tiempo. Su programación se basa en C++. A lo largo del tiempo han ido apareciendo versiones mejoradas de este producto, siendo las más actual la quinta.

- **GameMaker:** [5] Es una plataforma muy conocida y está basada en un lenguaje de programación interpretado y un SDK para desarrollar videojuegos. Es un programa muy sencillo y usado por personas que se inician en este mundo.

  Los tres aquí mostrados son los más comunes actualmente en el mundo de los videojuegos. Existen muchos más, cada uno con su cualidad específica. Algunos ejemplos también famosos son Cryengine, Ogre3D, Stencyl o UbiArt Framework, etc. Por otro lado, el desarrollo para el web apoyado en JavaScript (tanto en el cliente como en el servidor), Java y HTML5, que permite que desde el propio navegador se ejecute una escena está cobrando relevancia como demuestran APIs y *frameworks* como Web3D[3], WebGL[4], AJAX3D[5], Three.js[6], Processing.js[7].

2.4.2 Implementación de servicios red en juegos

La mayoría de los juegos multijugadores actuales utilizan la filosofía **P2P**[8] para hacerlos funcionar correctamente. En la cual no hay una entidad central y cada jugador controla el estado de su juego. En la filosofía P2P, cada jugador envía y recibe datos procedentes de todos los jugadores conectados, asumiendo que la información recibida es fiable y libre de intrusiones [6][7].

[2] Juegos o Aplicaciones FPS- First Person Shooters
[3] https://www.web3d.org/
[4] https://developer.mozilla.org/es/docs/Web/API/WebGL_API
[5] http://ajax3d.sourceforge.net/
[6] https://threejs.org/
[7] http://processingjs.org/
[8] Arquitectura P2P - Peer-to-peer

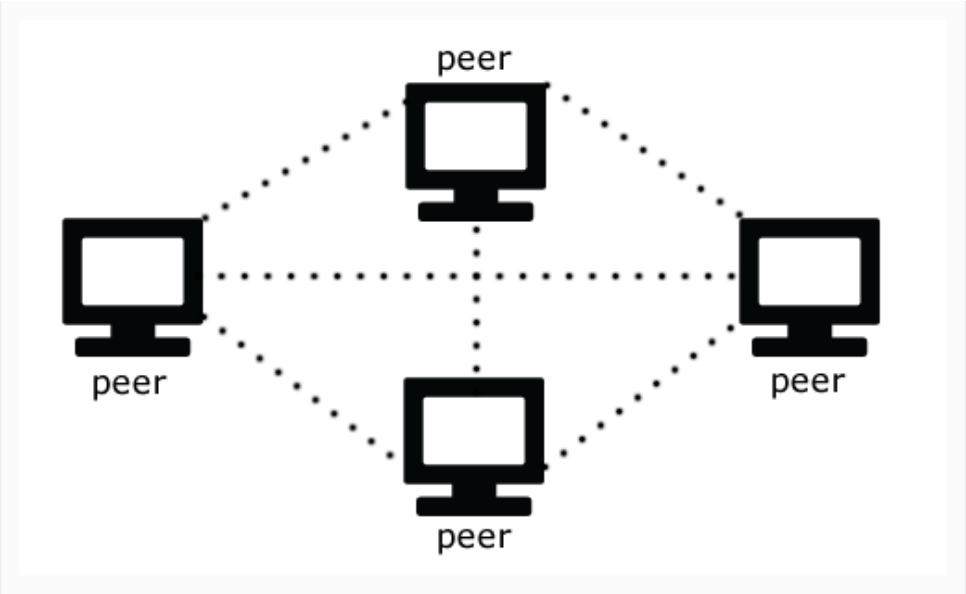

Figura 1. Implantación de la arquitectura P2P

Por lo tanto, la mayoría de los primeros juegos MMORPG[9] de desarrollaban siguiendo esta filosofía. A continuación, en la Figura 2 se muestra un claro ejemplo de cómo funciona esta arquitectura a la hora de aplicarla a un juego.

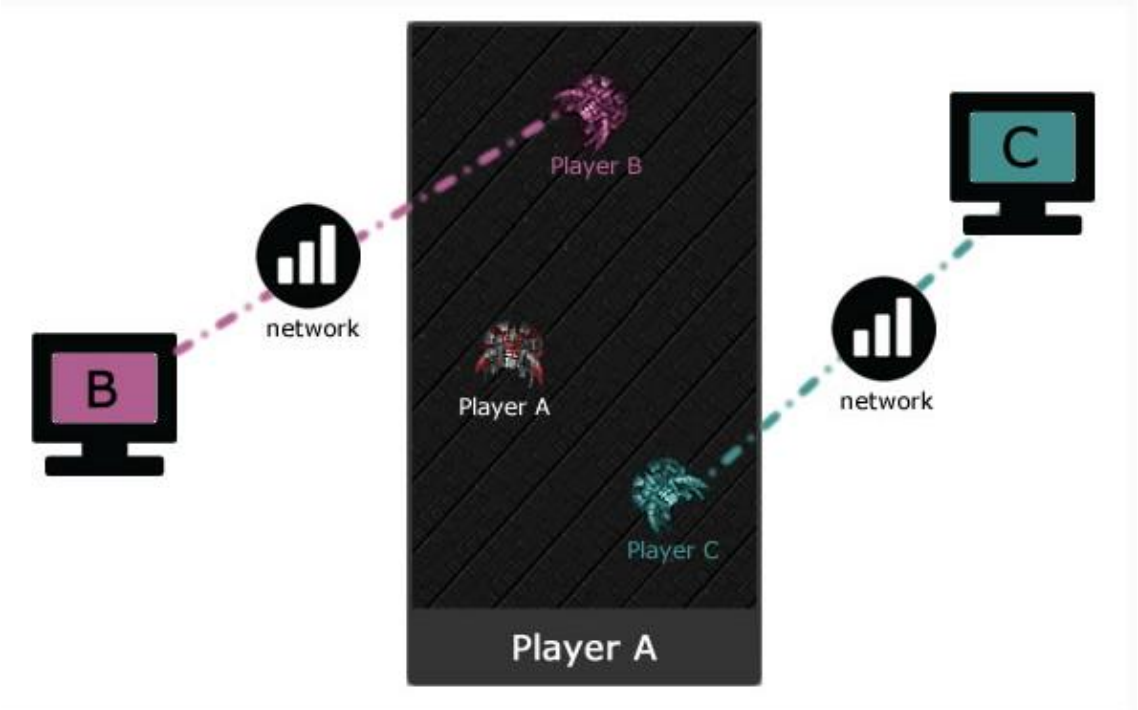

Figura 2. Juego funcionando en arquitectura P2P

[9] Juegos MMORPG - Massively Multiplayer Online Role-Playing Game

En la Figura 2, el jugador A recibe información del jugador C y B los cuales envían los datos relativos a su estado. El jugador B, por ejemplo, recibirá datos del jugador A y C simultáneamente, por lo tanto, se evita la necesidad de un servidor central que gestione todo el proceso.

Esta filosofía P2P también se basa en el uso de RPC (*Remote Procedure Call*) para el envío de la información entre los jugadores. Entiéndase RPC como un proceso de comunicación que permite a un programa ejecutar una subrutina o procedimiento en otro ordenador perteneciente a la misma red compartida sin la necesidad de especificar explícitamente en el código los detalles de esa interacción remota. Es interesante saber que en este proyecto también se utilizará RPC para interactuar con los jugadores a pesar de que, en el caso del juego desarrollado, se utilizará una arquitectura cliente-servidor. A continuación, en la Figura 3 se muestra un ejemplo de la arquitectura cliente-servidor desarrollada en el proyecto, de la cual se hablará más adelante.

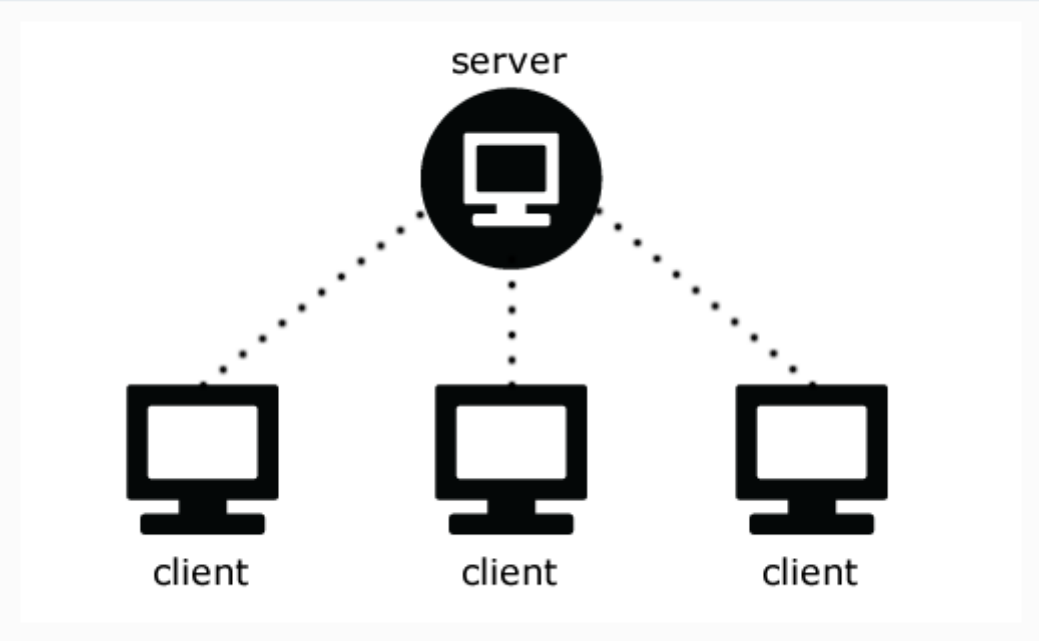

Figura 3. Arquitectura cliente-servidor

La creación de juegos MMORPG puede realizarse también mediante la tecnología **XML-RPC** [8][9] (*Markup Language Remote Procedure Call*). El cual es básicamente un protocolo basado en XML que se comunica mediante procedimientos remotos en un equipo remoto. Mediante esta tecnología se puede construir una base sólida para la construcción de un sistema cliente-

servidor que puede funcionar para un MMORPG. A niveles bajos, esta tecnología utiliza HTTP para comunicación.

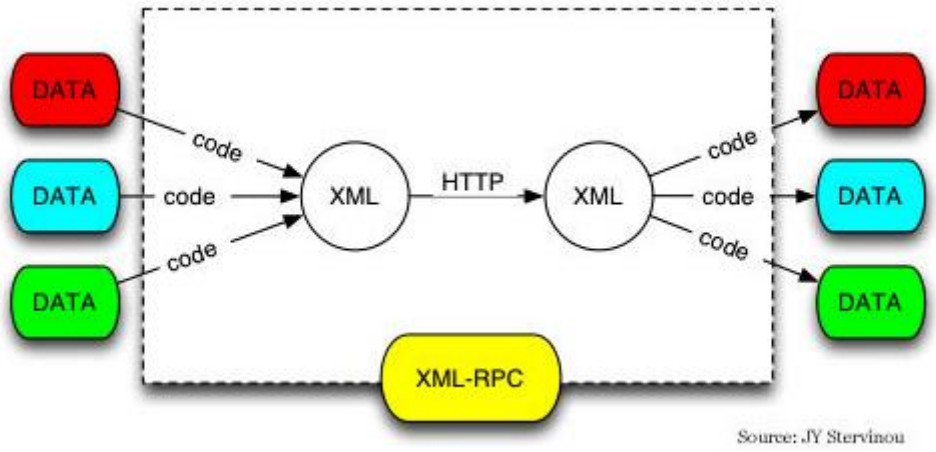

Figura 4. Tecnología XML-RPC

Es de especial interés saber que existen APIs que se sirven del uso de **WebSockets** y **HTTP** para la señalización, ya que en este proyecto se hace uso de una API que realiza dicha función para poder realizar implementación del *streaming* en Unity3D. Además, se hace uso de servidores ***Session Traversal Utilities for NAT***[10] (STUN) y ***Traversal Using Relays around NAT***[11] (TURN) los cuales en el proyecto se han usado los públicos de Google) para evitar problemas de conectividad debido a NAT y *firewalls* propios de usuarios domésticos. Para el envío de los flujos de audio y vídeo se utiliza *Stream Control Transmission Protocol*[12] (SCTP) y *Secure Real-time Transport Protocol*[13] (SRTP).

SCTP [10] es un protocolo que opera en la capa de transporte cuyo rol es similar al que realizan los protocolos UDP y TCP. Se podría decir que es una mezcla entre los dos, por un lado, es orientado a conexión como TCP y por otro, es orientado a mensajes como UDP.

[10] https://tools.ietf.org/html/rfc5389
[11] https://tools.ietf.org/html/rfc5766
[12] https://tools.ietf.org/html/rfc5061
[13] https://tools.ietf.org/html/rfc3711

SRTP [11] es la versión segura del protocolo RTP. SRTP proporciona encriptación y autenticación para evitar ataques de denegación de servidor. SRTP es la versión de RTP usada para WebRTC.

En el caso de Unity3D, la innovación en este terreno no para de crecer. Sin ir más lejos, la API que aquí en este proyecto ha sido objeto de estudio está siendo reemplazada en la actualidad por otra nueva llamada ***Multiplay*** [12].

Multiplay [13] será una evolución lógica de su predecesor, y mantendrá en cierto modo la filosofía de *Unity Networking*. Son pocos los datos aún de esta nueva versión, ya que actualmente se encuentra en su versión *Alpha*, pero sí va apareciendo información por parte de Unity3D que adelantan nuevos datos sobre cómo será esta nueva actualización. Uno de los datos más relevantes es la mejora de la opción "servidor dedicado": De hecho, ya existen apuestas bastante extendidas por el uso de plataformas como *DarkRift* que sobre Unity3D incorpora capacidades de creación de servidores dedicados con funcionalidad añadida para juegos FPS o *FirstPersonShooter* y MMORPG, que suelen ser los que más demandas a nivel de red tienen.

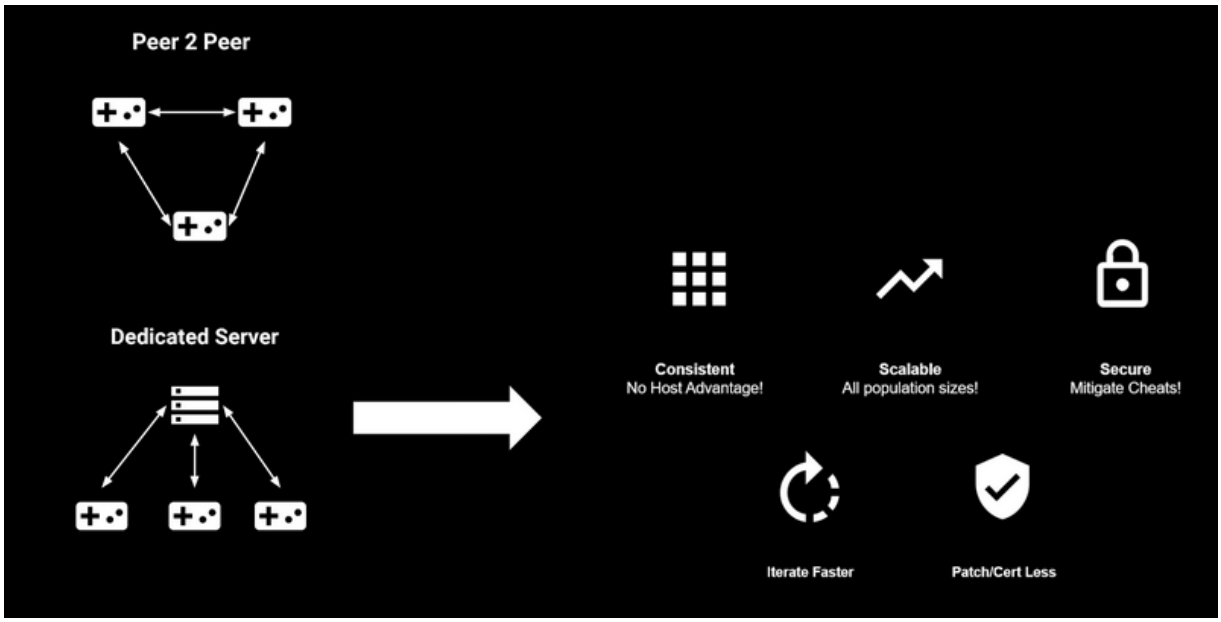

Figura 5. Arquitecturas de Multiplay

La nueva versión traerá un sistema de seguridad basado en códigos para evitar la presencia de jugadores maliciosos. Mejorará la escalabilidad de los jugadores y consistencia para aquellos juegos que requieran mayor consumo

de red. También una de las características más importantes es la reducción significativa del consumo de ancho de banda, ya que se pretende disminuir el número de información enviada, de manera que solo se envíe aquella de vital importancia al finalizar cada fotograma, de forma similar a cómo se realizan las descripciones 3D en los *profiles* de los códecs de video desarrollados bajo el estándar MPEG4-H.264.

No solo la API de Unity3D es la única solución multijugador existente. Se debe mencionar la existencia de ***SpatialOS***, una plataforma que funciona en la nube, algo muy a tener en cuenta debido al aumento de las tecnologías de procesado distribuido como las asociadas a *grid computing*, *cloud computing* y *fog computing*. Con la existencia de SpatialOS, las limitaciones que sufrían los juegos multijugador debido a la diversidad de los PC usados se solucionarían en gran medida.
Debido al uso de la nube de SpatialOS [14], se podría obtener un mayor número de jugadores activos, aumentar el tamaño de los mundos virtuales y así como el uso y desarrollo de técnicas de Inteligencia Artificial más avanzadas y con más requisitos computacionales. Algunas desarrolladoras de videojuegos se encuentran investigando esta tecnología para poder aplicarlas en sus futuros títulos, un ejemplo de ello sería *Project X* del estudio Automaton, el cual podrá en teoría albergar unos 1000 jugadores en un mismo escenario.

Una de las últimas novedades del sector de los videojuegos, y que a día de hoy cada vez más empresas están empezando a desarrollar, es la creación de juegos en *streaming*. Un ejemplo de ellos, y probablemente el más conocido recientemente es el **Stadia**.

Stadia [15], como se ha mencionado antes, es un servicio de suscripción para videojuegos en la nube desarrollado por Google. Haciendo uso de esta tecnología, los jugadores pueden conectarse de manera remota a través de un navegador y desde cualquier dispositivo (teléfonos móviles, Tablet, PC) a los centros de datos de Google con la capacidad de retransmitir el juego por *streaming* a 60 fotogramas por segundo, con soporte HDR y a una resolución de 4K. Google ha implementado este servicio mediante su navegador y a través de Youtube. El usuario deberá disponer de una buena conexión a internet para poder utilizar este servicio sin problemas.

2.4.3 Transmisión multimedia en tiempo real

En el mundo de los videojuegos es importante disponer de protocolos de transmisión en tiempo real para dotar a los juegos de comunicación directa entre sus usuarios. Algunos juegos incorporan estas funcionalidades para que sus jugadores no tengan la necesidad de usar otros programas específicos como *Skype, Discord, Team Speak*, etc. Existen varios protocolos que permiten el envío de datos a tiempo real de los cuales los más famosos son ***Real Time Transport Protocol***[14] (RTP) y ***Real Time Control Protocol***[15] (RTCP).

RTP es un protocolo de la capa de transporte que especifica la manera en la que los programas manejan la transmisión de datos multimedia a tiempo real sobre una red *unicast* o *multicast*. RTP no garantiza por sí mismo la entrega de los datos multimedia a su destino, pero sí gestiona la recepción de los datos. RTP combina su capa de datos con un protocolo de control llamado RTCP. Mediante RTCP, se monitoriza la red para captar pérdidas de paquetes y compensar el retardo que se pueda sufrir. RTP es un protocolo que funciona encima de UDP, y que cuyos protocolos de sesión pueden ser ***Session Initiation Protocol***[16] (SIP) o H.323.

El protocolo de sesión más utilizado es SIP, que permite la iniciación, modificación y finalización de sesiones interactivas donde los usuarios se intercambian datos multimedia. También es importante destacar el uso de **Session Description Protocol**[17] (SDP) el cuál se encarga de describir la sesión para el intercambio de datos entre dos puntos (básicamente una descripción de las capacidades de envío y recepción de cada punto). SDP típicamente se encapsula dentro de otro protocolo, siendo más común ir embebido dentro del protocolo SIP.

[14] https://tools.ietf.org/html/rfc3550
[15] https://tools.ietf.org/html/rfc3605
[16] https://tools.ietf.org/html/rfc3261
[17] https://tools.ietf.org/html/rfc4566

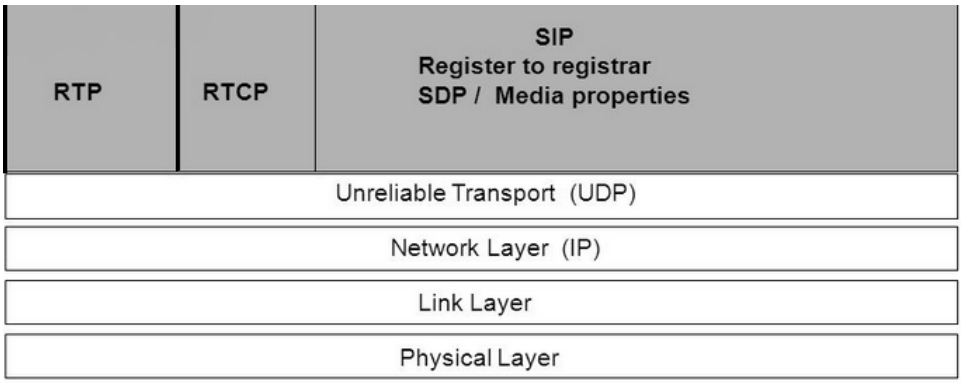

Figura 6. Torre de protocolos con RTP

2.4.3.1 WebRTC

WebRTC [16] es una API elaborada por W3C para permitir a las aplicaciones del navegador realizar llamadas de voz, chat de vídeo y el uso de archivos compartidos mediante una conexión P2P. De esta manera se elimina la necesidad de *software* cliente y *plugins*.

WebRTC es un trabajo en progreso con implementaciones avanzadas en los navegadores Chrome y Firefox. Está soportado por Google Chrome, Mozilla Firefox y Opera. La razón por la que se creó WebRTC es para combatir problemas de privacidad que aparecen al exponer capacidades y flujos locales. Este proyecto utilizará WebRTC como base para incluir los protocolos de tiempo real. En la Figura 7 se muestra los protocolos dispones en WebRTC.

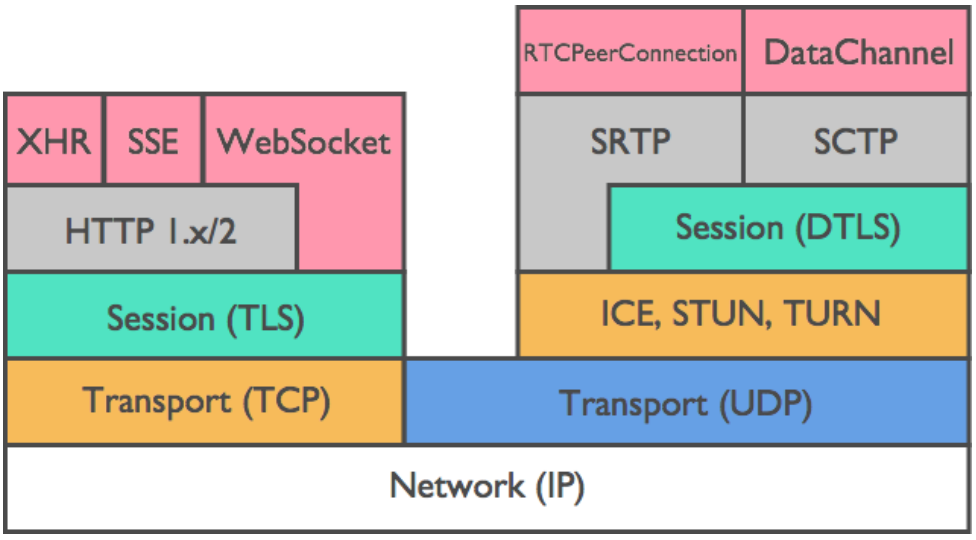

Figura 7. Protocolos de WebRTC

Para hacer funcionar WebRTC, se debe empezar por la señalización. La señalización es el proceso de coordinación de la comunicación en el que se acuerdan una serie de características, además permite tratar las siguientes funciones:

- Mensajes de control de sesión usados para abrir y cerrar la comunicación.
- Mensajes de error.
- El tipo de datos que vayan a ser usados en la comunicación.
- Claves para establecer comunicaciones seguras.
- Datos de red, tales como direcciones IP públicas y puertos.

En la Figura 7, aparecen los protocolos HTTP y WebSockets, los cuáles serán los utilizados para realizar la señalización en este proyecto. Ambos protocolos funcionan sobre TCP.

Otro punto importante es destacar el uso de una técnica llamada **Interactive Connectivity Establishment[18] (ICE),** junto con **Session Traversal Utilities for NAT (STUN)** y **Traversal Using Relays around NAT (TURN)** que aparecen en la Figura 7, y los cuales son transportados sobre UDP. ICE permite que WebRTC supere la complejidad que conlleva transportar datos fuera de una red privada. El trabajo de ICE es hallar el mejor camino para establecer la conexión entre dos usuarios que se comunican a través de la red externa. Funciona tanto para una conexión directa, como para una en la que las direcciones IP de los usuarios se encuentran detrás de un NAT. La Figura 8 esclarece el funcionamiento de ICE.

[18] https://tools.ietf.org/html/rfc8445

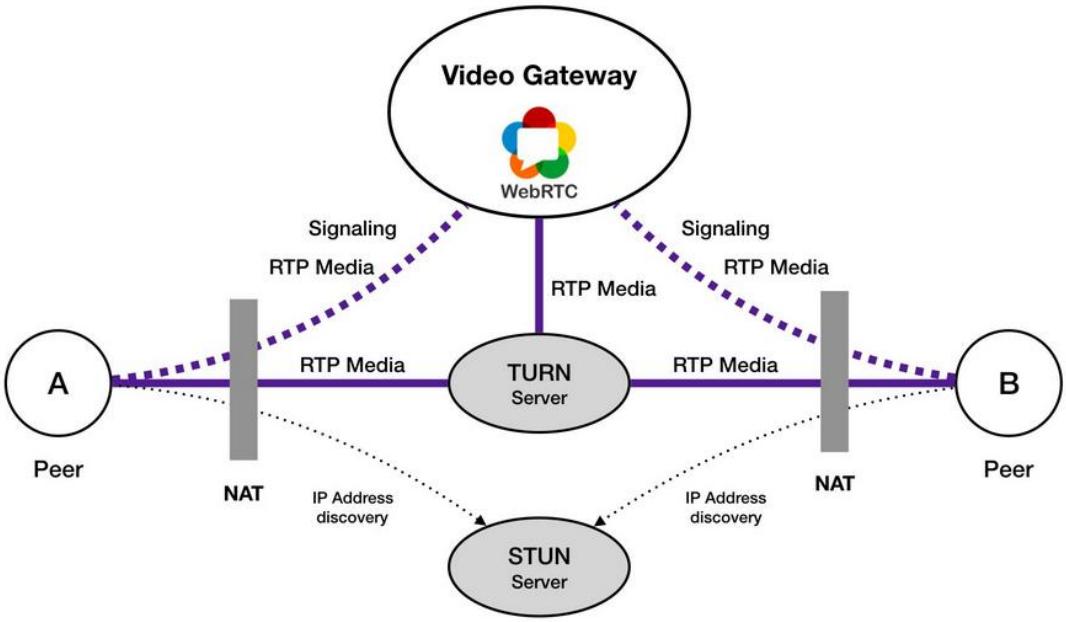

Figura 8. Uso de ICE en WebRTC

Servidor STUN: Mediante este servidor se permite a los clientes reconocer las direcciones IP públicas y el tipo de NAT que se está utilizando para poder establecer una llamada. En la mayoría de los casos, un servidor STUN es sólo utilizado durante el establecimiento de la conexión multimedia, y una vez que esta ha sido establecida, los datos fluirán directamente entre el *peer* y el *Video Gateway* (WebRTC). Como se puede ver en la Figura 8, el servidor STUN realiza el descubrimiento de las direcciones IP públicas involucradas en la comunicación, a pesar de que los equipos se encuentren tras un NAT. Posteriormente, una vez conocidas las direcciones se puede llevar a cabo la comunicación y el envío de los datos a través del *Video Gateway*.

Servidor TURN: Se utiliza cuando la comunicación de los usuarios mediante STUN falla (puede deberse a medidas de seguridad estrictas en la red). En estos casos, se tendrá que hacer uso de un servidor TURN para retransmitir el tráfico si la comunicación a través de la *Video Gateway* falla. Por lo tanto, a diferencia de lo que ocurría con el servidor STUN, los datos en este caso viajan directamente a través del servidor TURN.

2.5 Conceptos básicos

Hay una serie de conceptos que se deben conocer antes de entrar en profundidad a conocer cómo se ha realizado este proyecto. Estos conceptos se utilizarán mucho a lo largo de este documento, por lo tanto, se hace una breve presentación de los mismos

- ***GameObjects:*** [17] El *GameObject* (cuya traducción es "objeto de juego") es el concepto más importante en el Editor de Unity3D. Cada objeto del juego es un ***GameObject***, desde personajes y objetos coleccionables hasta luces, cámaras y efectos especiales. Sin embargo, un *GameObject* no puede hacer nada por sí mismo; se necesita dotarle de componentes antes de que pueda convertirse en un personaje, un entorno o un efecto especial.

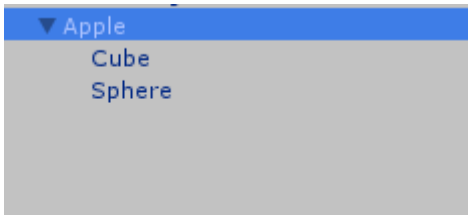

Figura 9. GameObjects Apple

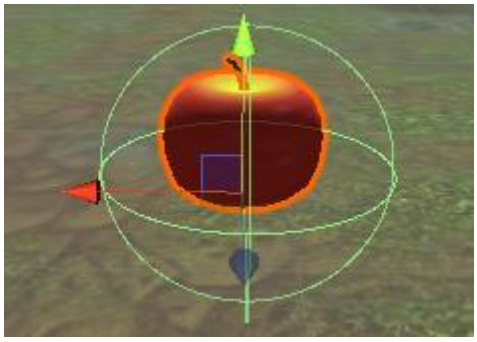

Figura 10. Representación del GameObject Apple en el Editor

- **Componentes:** [18] Son las piezas funcionales de cada ***GameObject***. El *GameObject* es un contenedor para muchos componentes distintos. Los componentes añaden funcionalidades al *GameObject*. Hay multitud de componentes en Unity3D y en este proyecto solo se han utilizado aquellos que han sido necesarios usar para hacer el juego funcional.

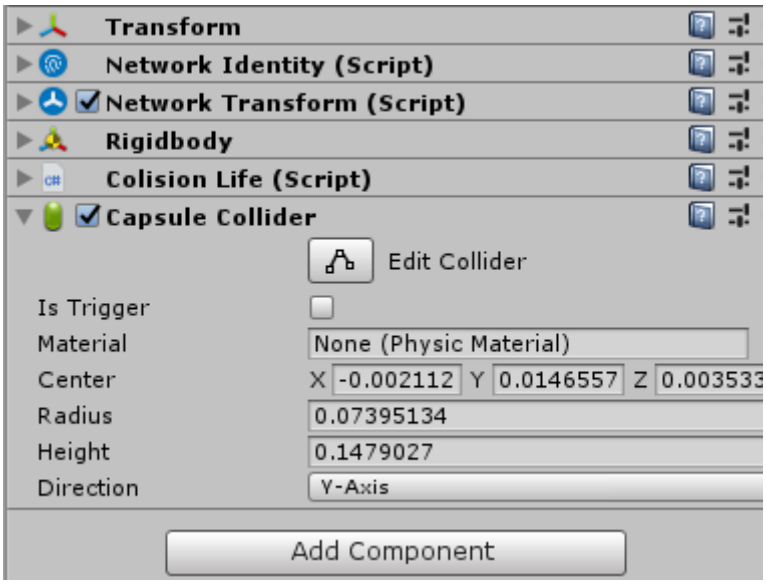

Figura 11. Componentes del GameObject Apple

Los componentes que aparecen en la Figura 11 (*Transform, Network Identity, Network Transform*…) serán explicados a lo largo de esta memoria.

- ***Asset Store:*** Es una plataforma que tiene Unity3D para comprar o descargar gratuitamente cualquier tipo de *GameObjects* o elementos que serán de utilidad para la creación de cualquier proyecto con Unity3D. Para poder descargar algo de la tienda es necesario tener una cuenta en Unity3D. La totalidad de los elementos que han sido obtenidos de la tienda han sido gratuitos para la realización de este proyecto.

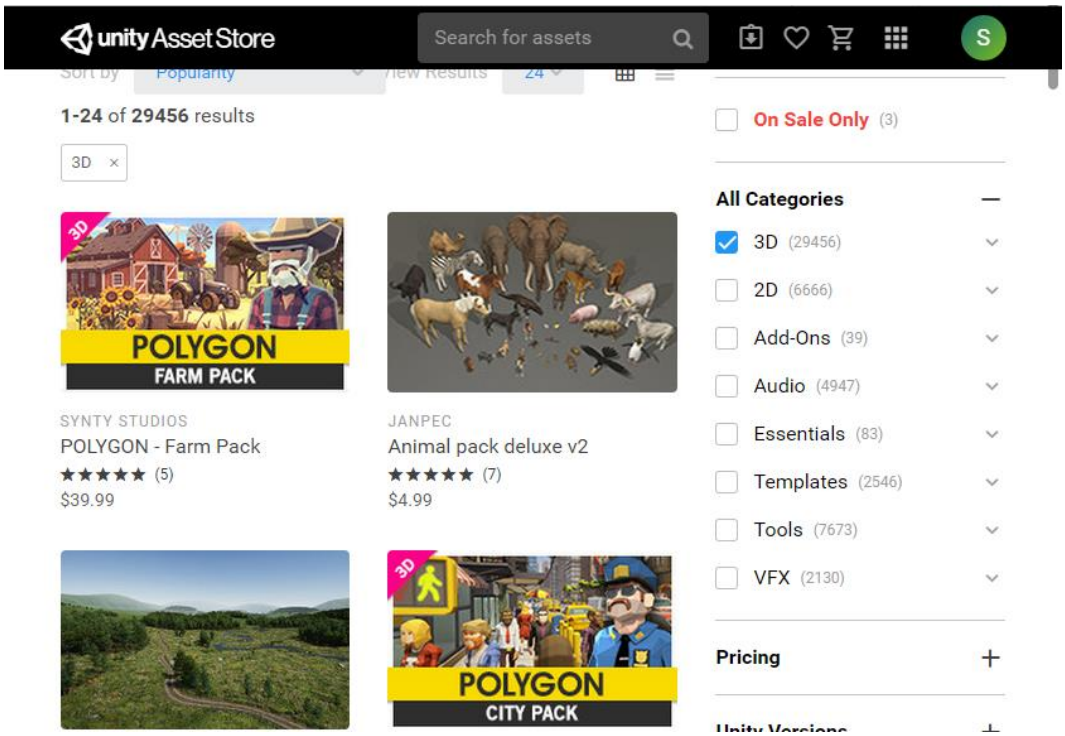

Figura 12. Asset Store

- ***Assets:*** Es el nombre que se la a los paquetes de cualquier tipo que se han obtenido desde la tienda o que ya trae Unity3D por defecto. Se han utilizado varios, como pueden ser *WebRTC Network*, *UJoystick*, *Network Lobby*, *Oculus Integration* y otros destinados a la creación del terreno del juego.

- **Oculus Go**: Son gafas de realidad virtual independientes. El dispositivo no se conecta a ningún ordenador y no requiere de un sistema de juego para funcionar. Solamente se trata de unas gafas con un sistema operativo Android integrado que no necesita estar conectado a un móvil.

3. Descripción del juego

En este capítulo se presentarán las partes de las que se componen el juego. Se dividirá el capítulo en apartados en los que se explicará qué hacer en cada pantalla para darle al lector una idea general del juego. Para terminar, se explicará cual es el objetivo final de la partida y como ganarla.

3.1 Ventana de inicio

Hay dos versiones disponibles del juego, una versión para PC y otra para Android. En la versión de PC, al entrar al acceso del juego aparecerá una ventana en donde se podrá configurar las opciones de vídeo más comunes; la resolución y la calidad gráfica. En la Figura 13 se muestra esa pantalla:

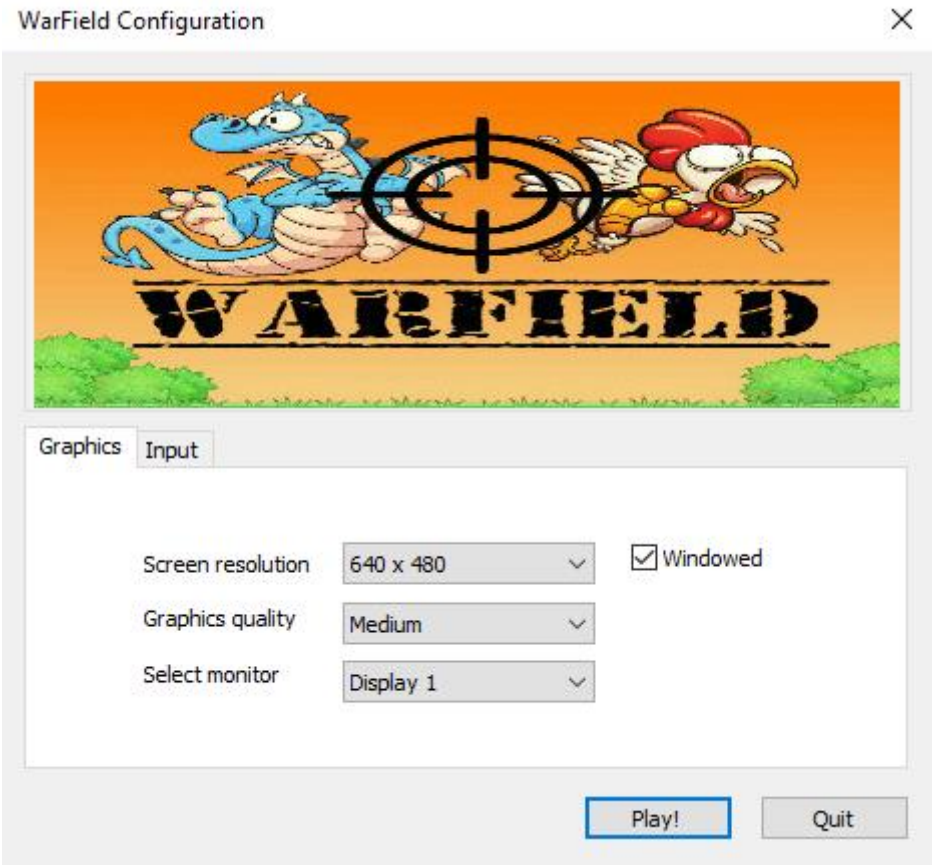

Figura 13. Pantalla de inicio del juego

Se podrá elegir la resolución de pantalla a la que se ejecutará el juego, así como la calidad de visualización. Esta calidad de visualización siempre será media, ya que se ha comprobado que es la configuración que permite un mejor rendimiento sin comprometer excesivamente la calidad.

En la versión Android no aparecerá esta pantalla de inicio, directamente se pasará al siguiente paso.

3.2 Ventana de configuración de salas o Lobby

En la Figura 14 se muestra el menú principal del juego, que se abrirá a continuación de la ventana de inicio en el caso de PC, o directamente al iniciar el juego en el caso de un dispositivo móvil Android.

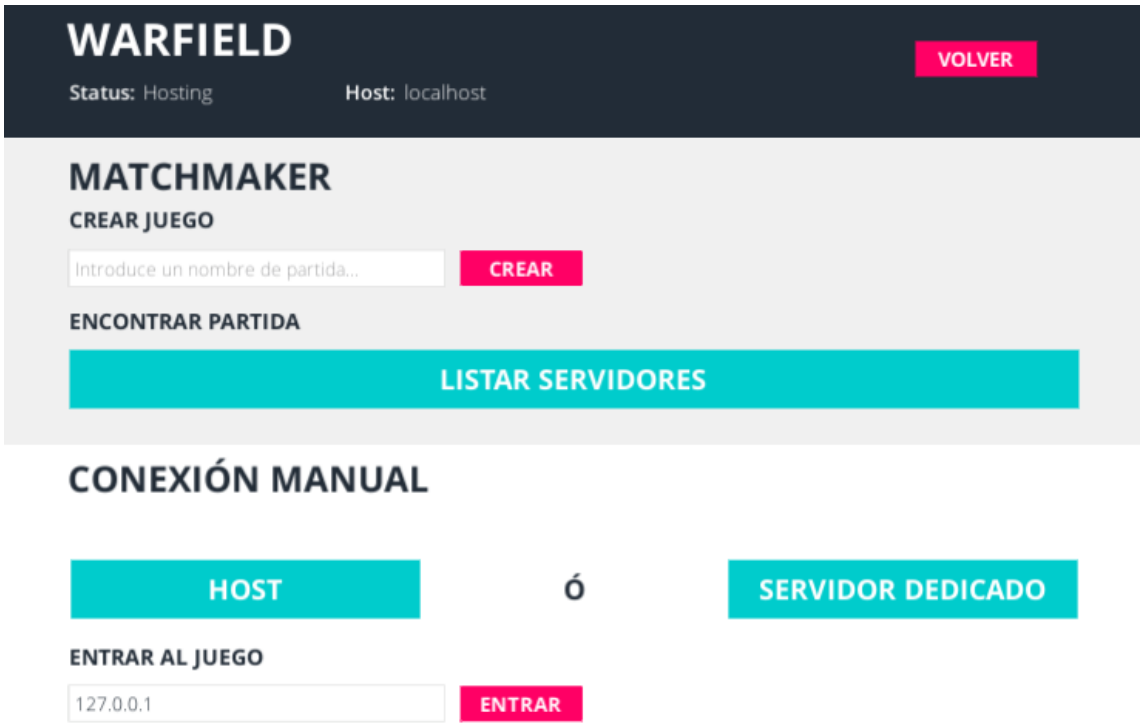

Figura 14. Pantalla inicial de configuración del juego

Esta sería la pantalla principal del juego, en ella tenemos las distintas opciones para crear una partida multijugador. Hay tres opciones de crear una partida, las cuáles se pueden dividir en dos tipos:

- Conexión manual.
- Mediante la nube de Unity3D o mediante un *MatchMaker*.

3.2.1 Conexión mediante un *MatchMaker*

Unity3D incluye servicios para que los jugadores jueguen con otros a través de Internet sin la necesidad de una dirección IP pública. Los usuarios pueden crear partidas, obtener listas de las partidas creadas; y unirse o salirse de las partidas. Cuando los jugadores juegan sobre internet, el tráfico de red viaja a través de un servidor de retransmisión o *relay* gestionado por Unity3D en la nube. Esto evita problemas con firewalls y NATS, permitiendo a los usuarios jugar desde cualquier lugar. Unity3D utiliza protocolos propietarios que implementan la misma funcionalidad o parecida a la ofrecida por los protocolos estandarizados por el IETF STURN, TURN y ICE.

La funcionalidad de *MatchMaking* puede ser utilizada usando un script especial llamado *NetworkMatch* en el namespace ***Unity3DEngine.Networking. Match***. En el proyecto solo ha sido necesario descargar un *asset* de la *Asset Store* llamado *Network Lobby*, el cual es de uso gratuito. Este *asset* ha sido posteriormente retocado para que pueda cumplir las necesidades del proyecto. Para hacer uso de esta característica, un jugador debe primeramente iniciar una partida. Para ello se debe poner un nombre a la partida que se va a crear para que los usuarios que recojan la lista de partidas sepan diferenciar cual es la que buscan.

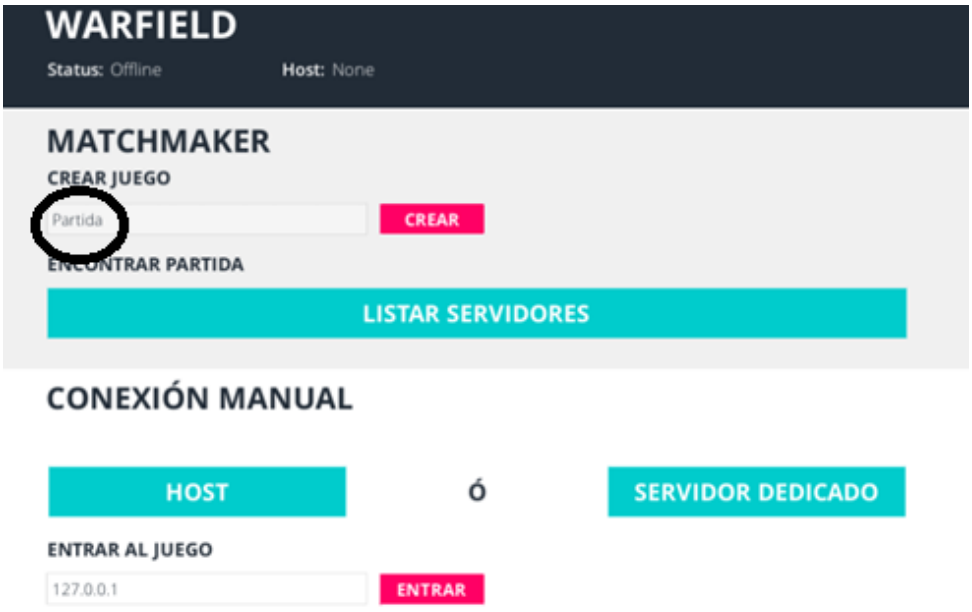

Figura 15. Creación de una partida mediante MatchMaker

Como se puede apreciar en la Figura 15, se le ha puesto el nombre “Partida” a la partida. Tan solo se le debe dar después a Crear, y aparecerá una ventana en la que el jugador pueda introducir su nombre y elegir el color de su avatar (Azul en la Figura 16).

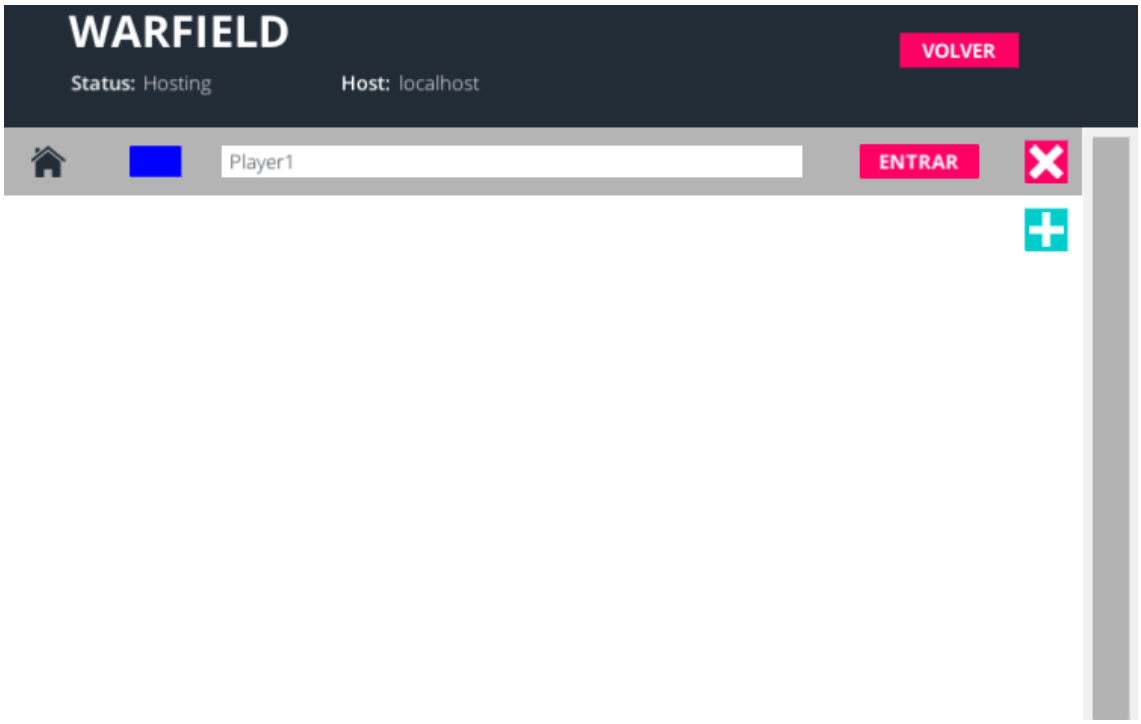

Figura 16. Ventana de partida

En el caso del otro jugador, se mostrará la ventana de la Figura 14 donde aparece el menú de inicio. En él se debe pulsar la tecla “Listar Servidores” y aparecerán las partidas que se encuentren creadas para el juego. En ocasiones puede ocurrir que un jugador cree una partida, y al cabo del rato decida cerrar su sala. Ese proceso de avisar al servidor de que la partida ha sido cancelada puede tardar unos segundos, y puede ser necesario esperar para que desaparezca la partida en la lista de servidores a los clientes.

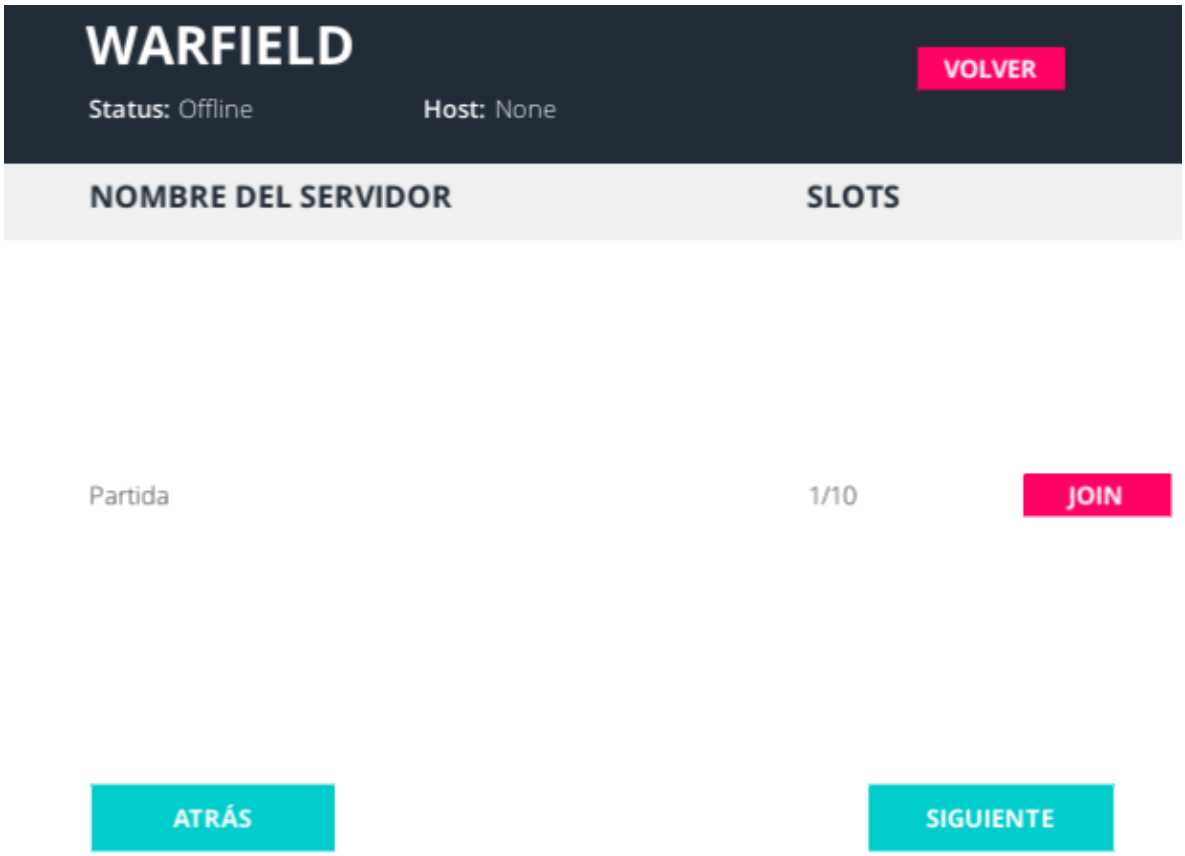

Figura 17. Lista de partidas abiertas

A continuación se elige la partida que se desee de la lista (Ver Figura 17) y se presiona en *Join*. En ese momento el cliente entrará a la sala del jugador que había creado la partida anteriormente.

3.2.2 Conexión de forma manual

Dentro del tipo de conexión existen dos formas: la conexión mediante *Host* o mediante un servidor dedicado.

- **Mediante *Host*:** el servidor tendrá la dirección IP del jugador que ha iniciado la partida como *Host*, los demás clientes entrarán a la partida introduciendo la dirección IP de dicho jugador. Es importante tener en cuenta que, si se inicia una partida con esta opción, el servidor se comportará también como otro jugador cualquiera de la partida.

- **Servidor dedicado:** En este caso, el equipo que se encarga de ser el servidor de la partida no será un jugador de la partida. Cuando un equipo se comporta como servidor y esté esperando que los jugadores se unan a su partida aparecerá una pantalla tal que así:

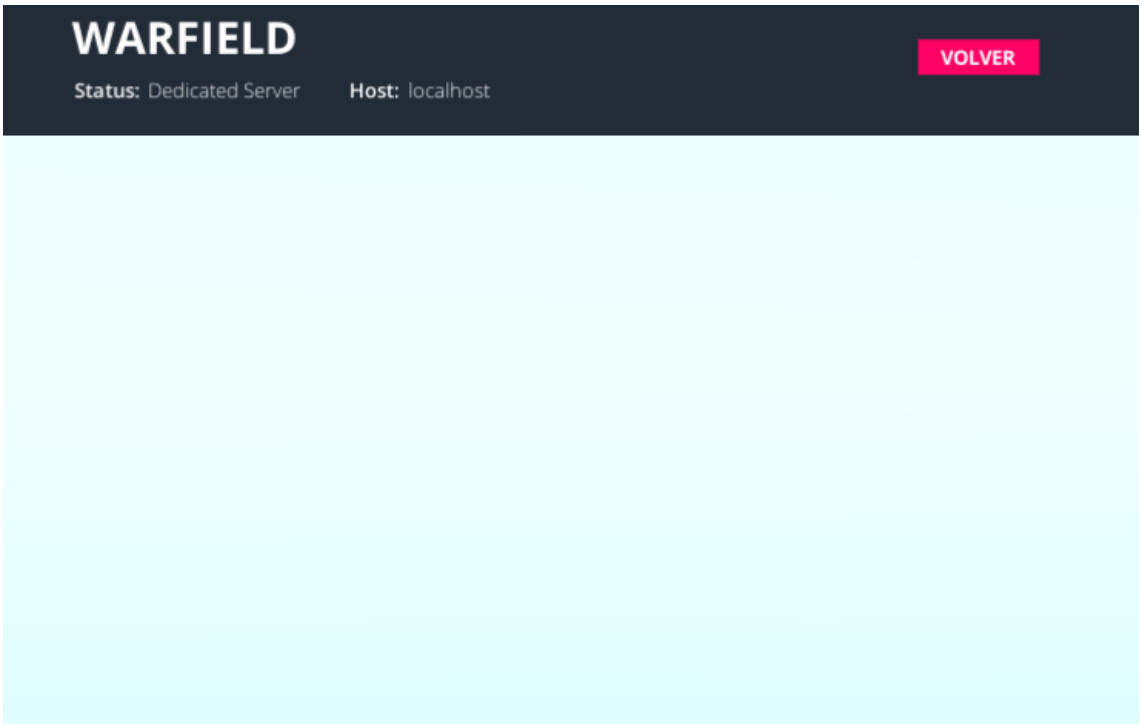

Figura 18. Pantalla de espera para un servidor dedicado.

Cuando todos los jugadores hayan entrado a la partida y hayan marcado la opción de que están preparados, iniciará la cuenta atrás para iniciar la partida. Cuando la partida haya iniciado, aparecerá la pantalla mostrada en la Figura 19, para el equipo que esté comportándose como servidor dedicado.

Figura 19. Pantalla de juego en el servidor dedicado.

No se puede tener dos servidores dedicados con la misma dirección IP. Si un servidor dedicado intenta crear partida cuando ya hay otro con su misma dirección IP creado le aparecerá un error al segundo servidor indicando que ya hay uno creado.

3.2 Escenario

Una vez que los jugadores hayan puesto su nombre y elegido un color asociado, aparecerá un contador en ambas pantallas indicando el tiempo de espera para el inicio de la partida, como el mostrado en la Figura 20.

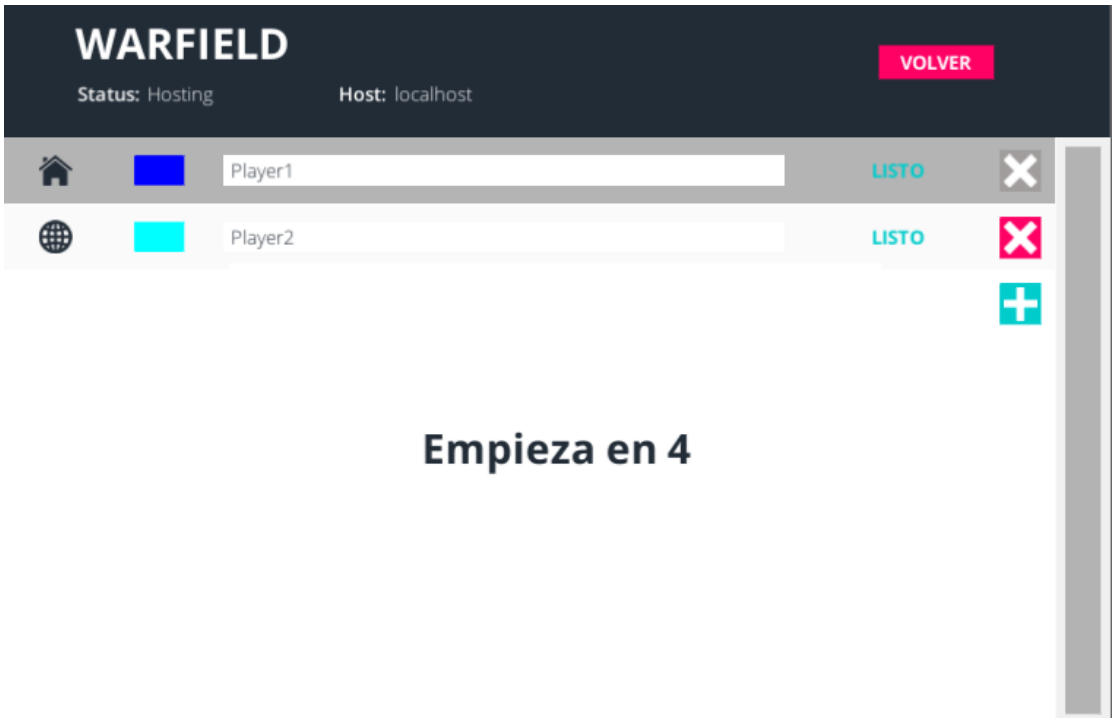

Figura 20. Temporizador de inicio de partida

Una vez terminada la cuenta atrás, el servidor se encargará de renderizar la escena para que esta sea posteriormente cargada en todos los clientes. Este proceso crea una carga notable en las funciones relacionadas con la API de *Networking* de Unity3D, la cual se verá en el capítulo de pruebas del presente documento.

Una vez renderizada la escena en todos los equipos, la partida habrá empezado, y cada jugador será colocado aleatoriamente en una de las posiciones programadas en el mapa como puntos de inicio.

El avatar será del color que el jugador haya seleccionado en la pantalla de partida, y su nombre, aunque él mismo no pueda verlo, aparecerá en la parte superior de su barra de salud en la cabeza del avatar. A continuación, en la Figura 21, se muestra cómo se vería el juego.

Figura 21. Vista del juego al iniciar partida

En la Figura 21 se puede ver una captura del juego una vez que ha iniciado la partida. En el momento de iniciar la partida los primeros enemigos habrán aparecido, avatares voladores, y empezarán a seguir al jugador más cercano que encuentren. Aparecerán nuevas oleadas de enemigos periódicamente en un periodo de entre 30 y 40 segundos.

3.3 Objetivo

Como se puede ver en la Figura 21, en la parte superior hay un número que indica la puntuación del jugador, el número de vidas y un temporizador. El objetivo del juego es sobrevivir a las oleadas de enemigos y obtener la máxima puntuación de todos. El límite de tiempo es de 3 minutos, durante ese periodo los jugadores deben luchar por conseguir el mayor número de puntos e intentar no perder las cinco vidas. A continuación, se describe en detalle las reglas de funcionamiento del juego:

- En el caso de que un jugador pierda todas las vidas antes de finalizar los tres minutos, aparecerá un mensaje de derrota, y este jugador no podrá mover ya su avatar.
- En el caso de que, por ejemplo, haya 5 jugadores y 4 de ellos hayan perdido todas las vidas, terminará la partida independientemente de que no hayan pasado los 3 minutos. Ganará el jugador que haya sobrevivido a las oleadas, independientemente de los puntos que tenga.
- En el caso de que hayan pasado los 3 minutos de partida, y hayan quedado 2 jugadores vivos, ganará aquel que tenga el mayor número de puntos.
- No hay penalización de puntos por cada vez que un jugador pierda una vida.
- Para ganar puntos los jugadores deberán quitarle vida a un enemigo, cada vez que un jugador dispare exitosamente a un enemigo recibirá 10 puntos.
- Los jugadores pueden matarse entre sí, pero no se sumará puntos como en el caso de los enemigos. Lo que se consigue al matar a un juego enemigos es quitarle una vida y tener, por lo tanto, más posibilidades de ganar la partida.
- A lo largo del escenario aparecerán manzanas, que si el jugador las recoge sumará 50 puntos de vida a control de salud.

4. Fase de desarrollo del videojuego

En este capítulo se explicará cómo se ha desarrollado desde cero el videojuego, empezado a utilizar el IDE asociado a Unity3D. para lo cual se describen inicialmente los componentes básicos que se han empleado, y posteriormente se explicará su uso en el proyecto.

4.1 Primeros pasos con Unity3D

La versión de Unity3D que se ha utilizado ha sido la **2018.2.6f1 (64-bit).** Se ha escogido esta versión porque es reciente e incorpora suficientes componentes de la API de Networking. Es importante recordar, como ya se había comentado en el capítulo 1 de esta memoria, que las nuevas versiones ya no incorporarán la API de Networking que se utiliza en este proyecto. La nueva API que la sustituirá se llama *Multiplay*, y aunque parece que será muy similar a su antecesora, cambiará notablemente la forma de implementarla.

En primer lugar, cuando se inicia el programa se procederá a crear un proyecto nuevo. Se deberá poner el nombre que se le quiere dar al juego.

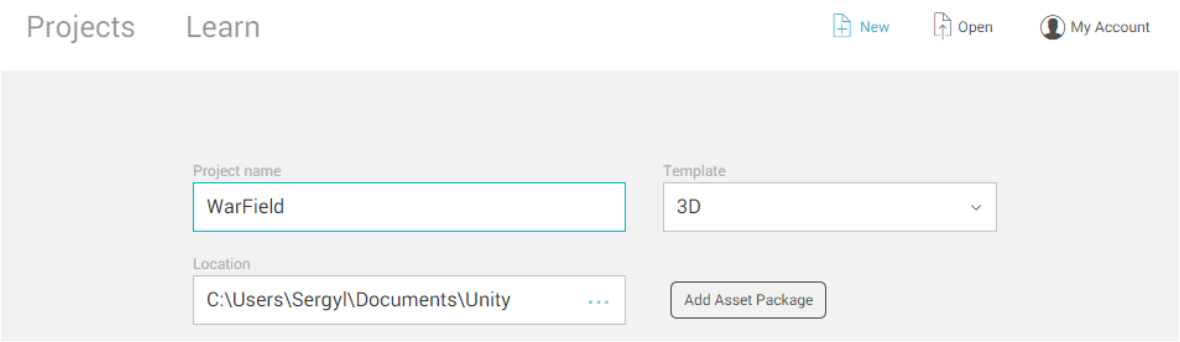

Figura 22. Ventana de creación de un proyecto

Una vez creado el proyecto, aparecerá la zona de trabajo de Unity3D en donde se procederá a la creación del juego. Mediante la opción *Layout* en la parte superior derecha se puede cambiar la manera de visualizar el entorno de trabajo. Por motivos de comodidad se ha elegido la opción "*2 by 3"*. A continuación, se muestra una imagen de dicha zona:

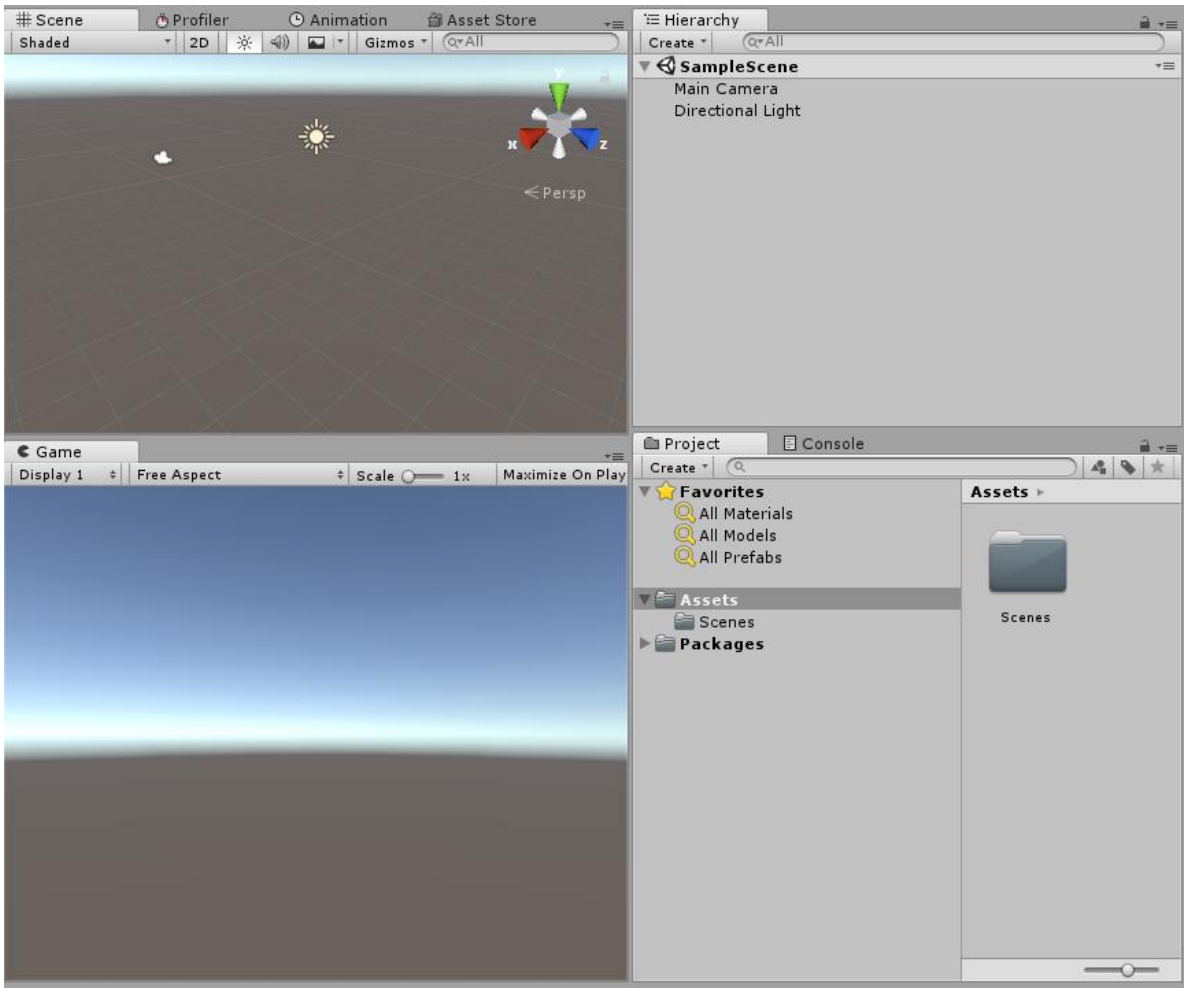

Figura 23. Zona de trabajo de Unity3D

4.2 Diseño del escenario

Lo primera fase es la de la realización del diseño del escenario del juego. En primer lugar, se procede a crear una escena nueva dentro del entorno de desarrollo la cual se llamará **Game**. En ella se diseñará un terreno medieval inventado. El primer paso es crear en la escena un *GameObject* llamado *Terrain*, el cuál tras su creación aparecerá de esta forma dentro la de escena:

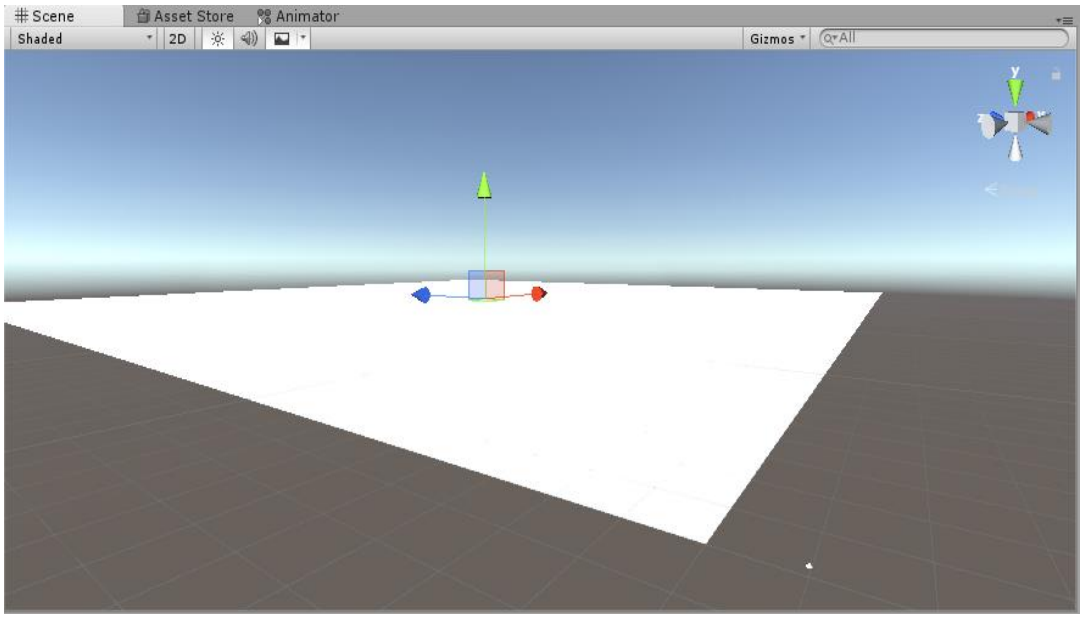

Figura 24. GameObject Terrain

Posteriormente, se tendrá que incluir las texturas correspondientes a las hierbas, las montañas, la nieve y cualquier efecto decorativo que se desee implementar dentro del terreno. Las texturas se pueden descargar a través de los *assets* que vienen ya hechos en la plataforma de ventas de Unity3D, llamada Asset Store.

El componente *Terrain* viene con tres componentes por defecto cuando este es creado. En la Figura 25 siguiente se muestran dichos componentes:

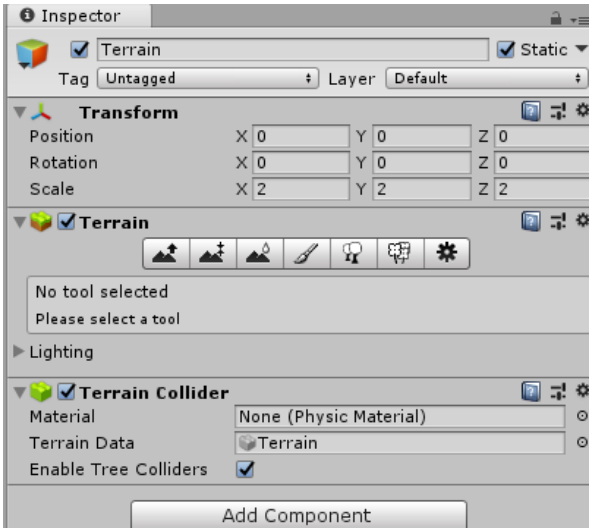

Figura 25. Componentes del GameObject Terrain

- El componente *Transform* establece la posición, rotación y tamaño del elemento en blanco que aparece en la Figura 25. Este componente lo poseen todos los *GameObjects* del juego.
- El componente Terrain permite crear montañas, desniveles, suavizar el terreno, pintar texturas o colocar árboles y hierba en el terreno.
- El componente *Terrain Collider* se encarga de gestionar el tema de las colisiones para el escenario. De esta manera, los jugadores no se caerán al vacío.

El componente Terrain será esencial para la elaboración del mapeado, las opciones que han sido utilizadas son las siguientes:

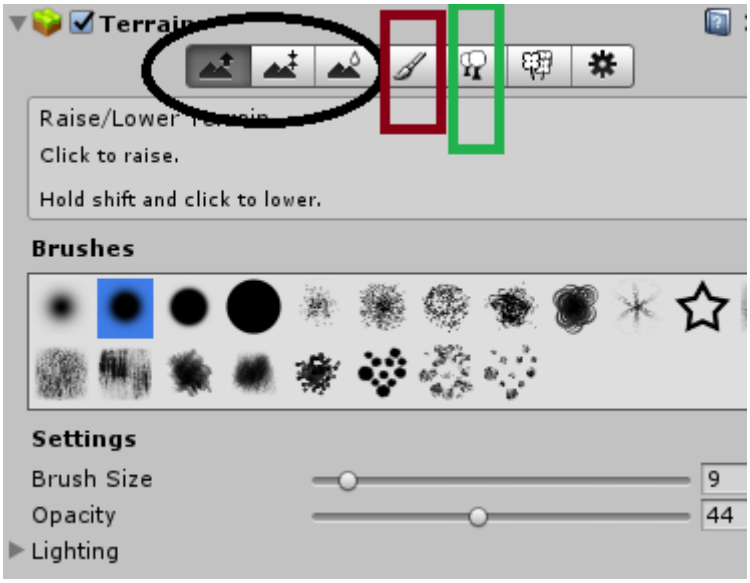

Figura 26. Características del componente Terrain

- Las opciones redondeadas con un círculo negro en la Figura 26 sirven para elevar el terreno (crear montañas, colinas, laderas), crear boquetes y suavizar el terreno para que no sea muy puntiagudo.
- La opción marcada con un cuadrado de color marrón permite pintar las texturas en el escenario. Para pintar una textura es necesario importarla previamente en el cuadro que pone Textures. En la Figura 27 se muestra las texturas que han sido utilizadas.

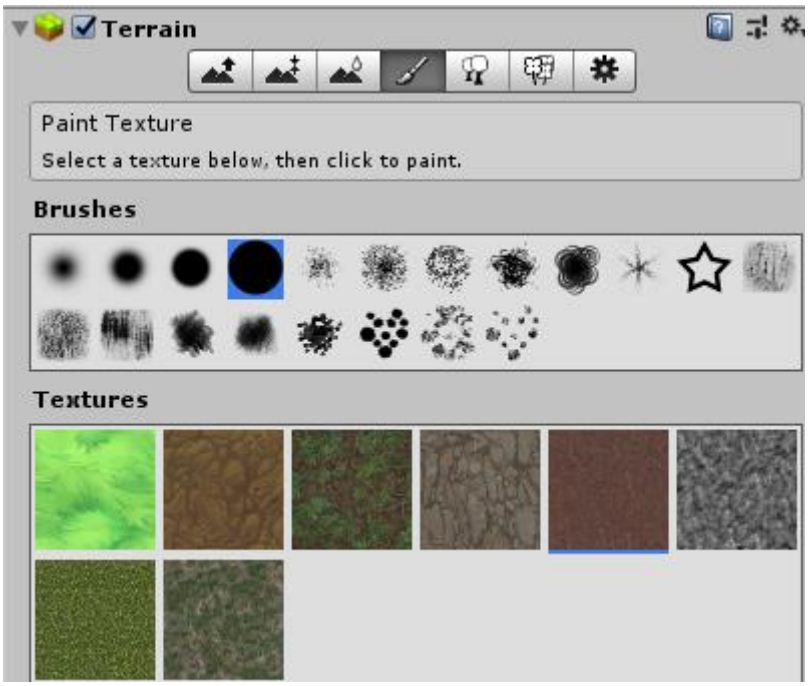

Figura 27. Texturas importadas para el terreno

Las texturas han sido descargadas a través de la plataforma Asset Store. Una vez descargadas las texturas, mediante la brocha que proporciona Unity3D se va pintando sobre el escenario hasta conseguir el resultado deseado.

Por último, se han añadidos árboles. Dichos árboles no han sido de descargados de la Asset Store, sino importados de los propios modelos que tiene Unity3D internamente. Para importar los modelos se debe ir a pestaña *Assets* de la parte superior izquierda, luego se debe pinchar en *Import Assets* y por último seleccionar que tipo de elementos se quiere importar.

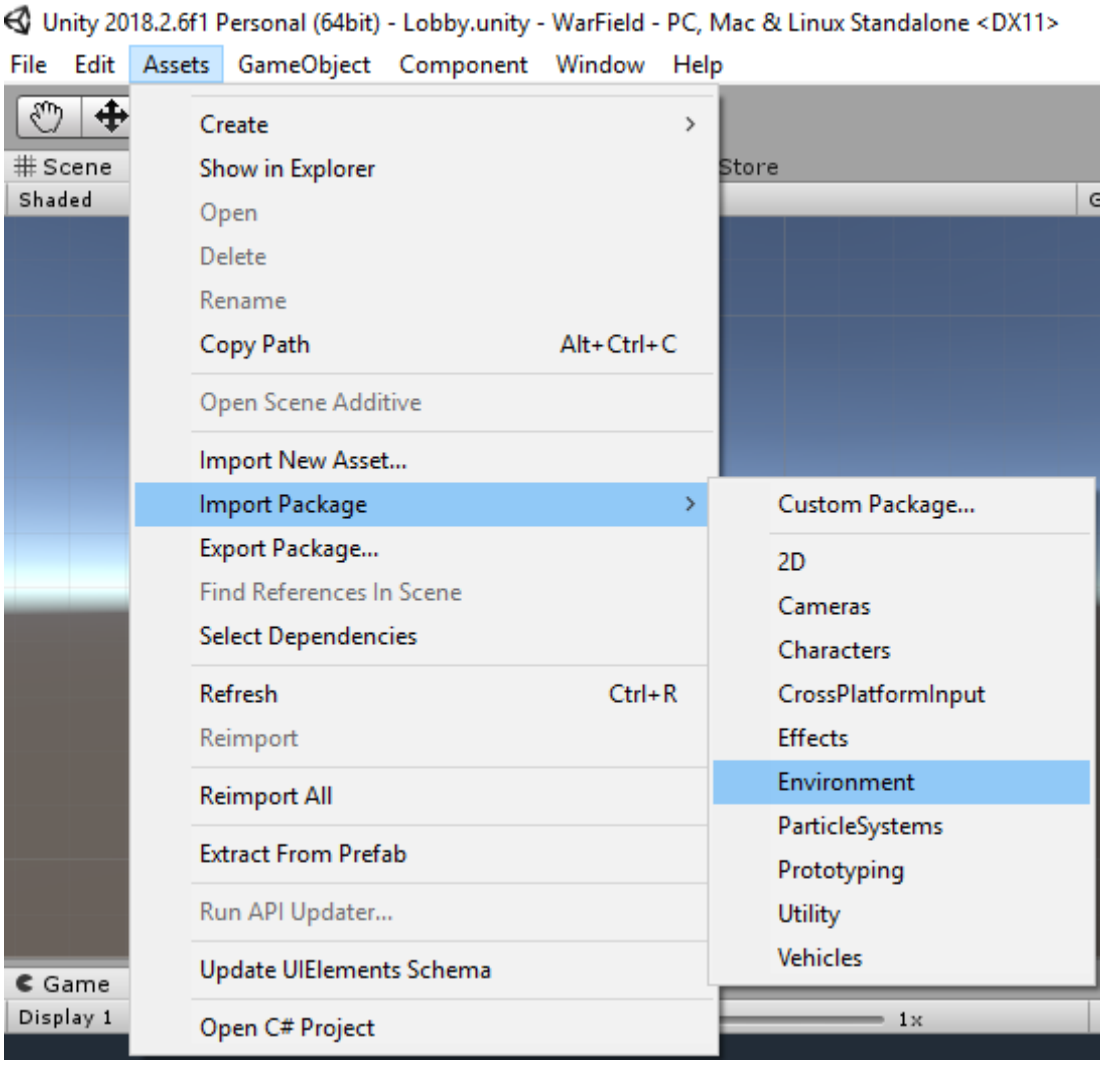

Figura 28. Importar paquetes de Unity3D

Una vez que se han importado los árboles, el siguiente paso es pintarlos sobre el escenario. Para ello hay que volver al componente *Terrain* y añadir los árboles que se desee pintar a la lista. Posteriormente, de la misma forma que se ha hecho con las texturas, se debe pintar con la brocha los árboles en el escenario.

Se debe tener en cuenta que cuanto mayor sea la cantidad de árboles en la escena, mayor carga gráfica tendrá que soportar la máquina que ejecute el juego. Teniendo en cuenta que el juego también va a funcionar en un dispositivo móvil, no se ha colocado una cantidad de árboles excesiva para no poner en peligro el rendimiento del juego en estos dispositivos.

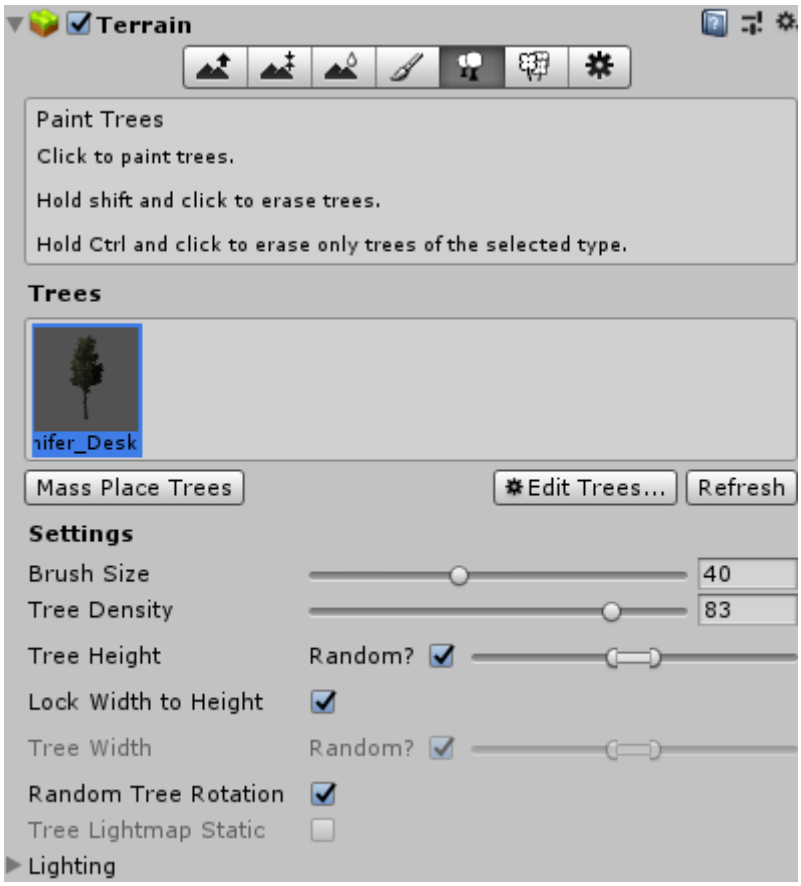

Figura 29. Opción para el pintado de los árboles

Además de controlar la cantidad de árboles en la escena, para mejorar el rendimiento del juego aún más, se ha decidido prescindir de ciertas opciones gráficas de los árboles como la calidad de las sombras.

Hay más opciones gráficas que se han modificado en favor de mejorar el rendimiento del juego lo máximo posible. Así como se muestra en la Figura 30, se han hecho uso de las características de LOD o *Levels of Deteails*, que modifican la calidad y/o complejidad de la representación gráfica en función de la distancia al punto de visas o el uso de *Billboards* o texturas que se mueven en función de la rotación del punto de vista. Dichas opciones aparecerán en apartados posteriores del presente capítulo.

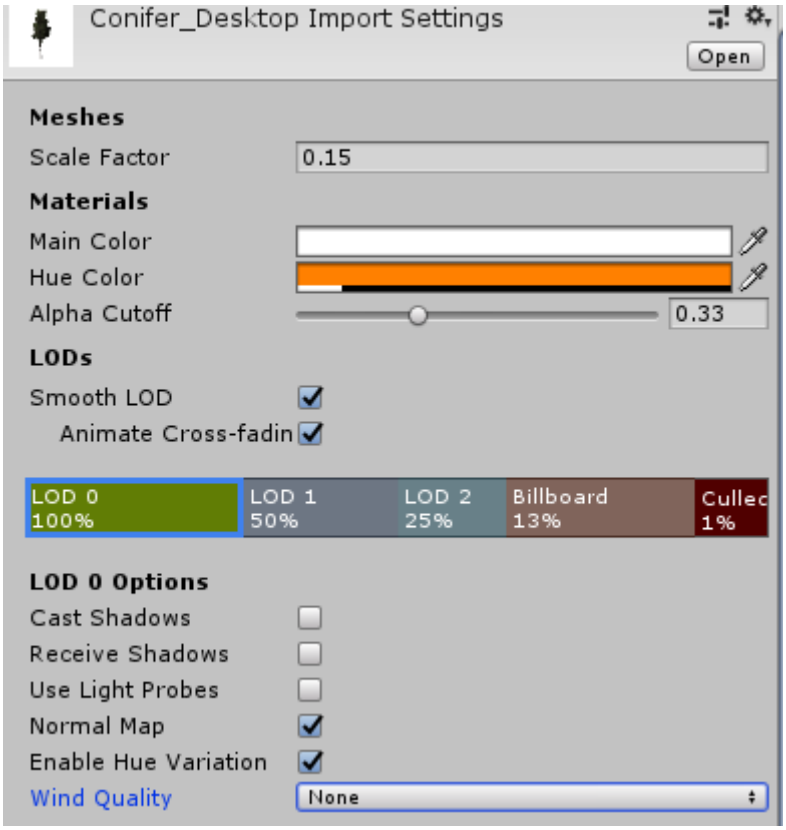

Figura 30. Características gráficas de los árboles

Aparte de todos los elementos mencionados hasta ahora, también se ha decido colocar un río en mitad del escenario para dividir el terreno y aumentar un poco la dificultad del juego. El río también ha sido un *asset* importado de Unity3D el cual venía dentro del mismo paquete que los árboles. Colocarlo es sencillo, es un *GameObject* como otro cualquiera al cual se le asocia un script que se encargará de realizar el efecto de oleaje.

Finalmente, se ha añadido componentes medievales con el fin de mejorar el aspecto del mapa, entre ellos se encuentran; un castillo, una torre derruida, un poblado y puentes. Mediante los puentes se puede cruzar el río.

Estas construcciones han sido descargadas de la *Asset Store* de manera gratuita y también se les ha reducido el tamaño máximo de las texturas originales para mejorar el rendimiento lo máximo posible. A continuación, se muestra imágenes (Figuras 31 y 32) del escenario creado desde varias perspectivas mediante las herramientas mencionadas anteriormente:

Figura 31. Escenario del juego con vista lateral

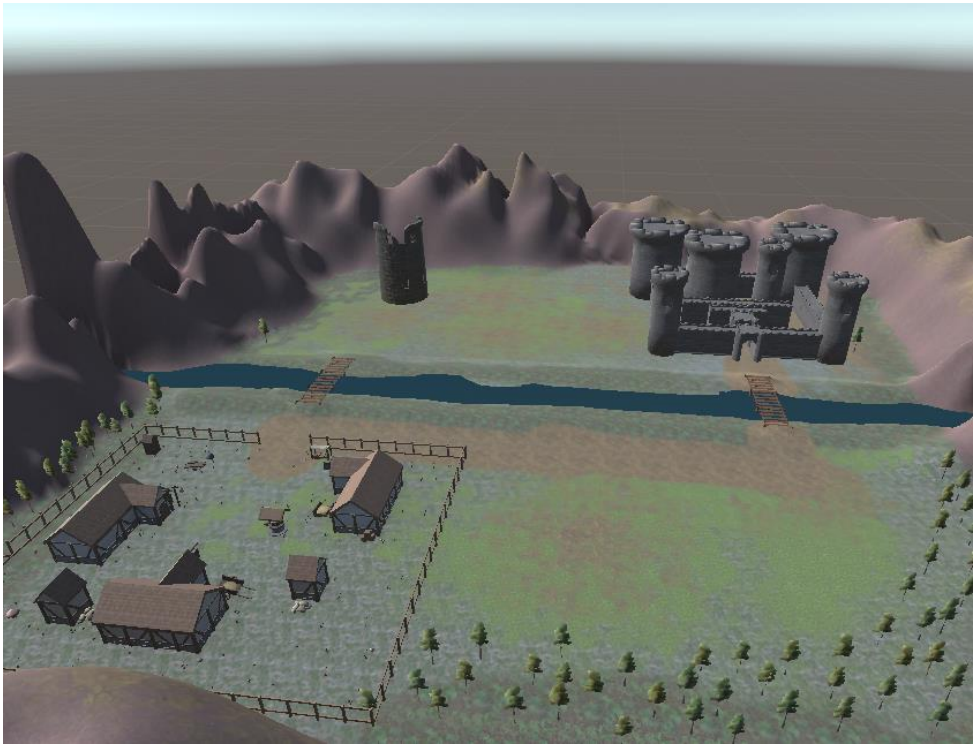

Figura 32. Escenario del juego con vista frontal

4.3 Colisiones en el escenario

Las colisiones son otro elemento imprescindible del juego y que han sido muy utilizados. Mediante las colisiones se consigue que los elementos detecten el momento en el que un *GameObject* choca contra otro elemento. Cuando no existían motores de videojuegos capaces de gestionar este comportamiento, la detección de colisiones era un problema bastante complicado para resolver ya que se tenía que realizar numerosos cálculos matriciales, así como el uso de modelos simplificados basados en hacer una descripción convexa que facilitara la detección de la colisión.

Con Unity3D se debe incluir un componente llamado *Collider* a los *GameObjects* que se desea dotar de masa. Dentro del escenario se han colocado paredes invisibles para que delimiten ciertas zonas del escenario.

Principalmente se delimita los extremos del mapa para que el jugador no pueda subir las montañas y salirse, además se limita también que el jugador pueda entrar al río, ya que le restaría realismo. En la Figura 33 se mostrará en color verde las paredes de colisiones colocadas en el escenario para conseguir estos fines.

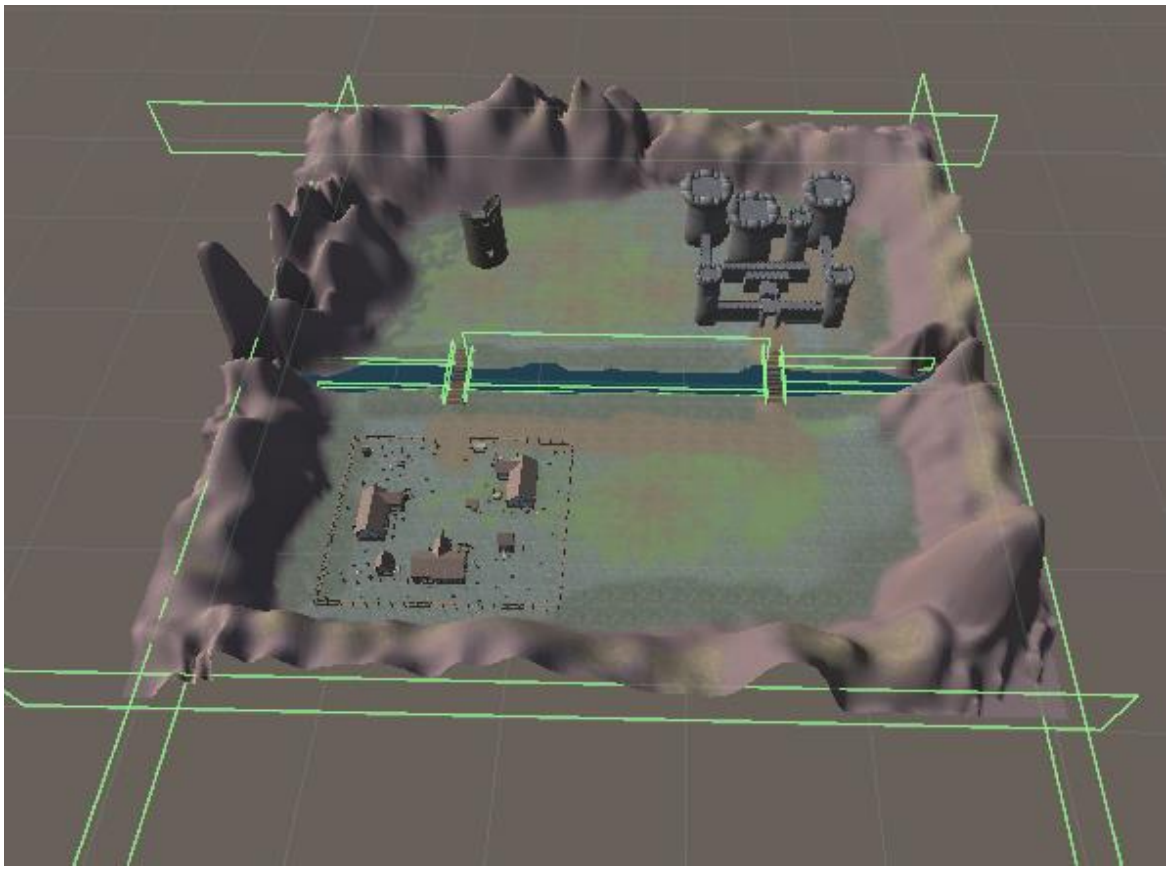

Figura 33. Colisiones en el escenario

4.4 Enemigos

Como se ha comentado en el capítulo de la descripción del juego, los jugadores tendrán que sobrevivir a varias oleadas de enemigos y conseguir los máximos puntos posibles para ganar la partida. Para conseguir estos puntos se debe quitar vida a cualquiera de los enemigos que vagan por el mapa. Hay tres tipos de enemigos, y cada tipo aparece en un lugar diferente del mapa (Ver Figuras 34, 35 y 36).

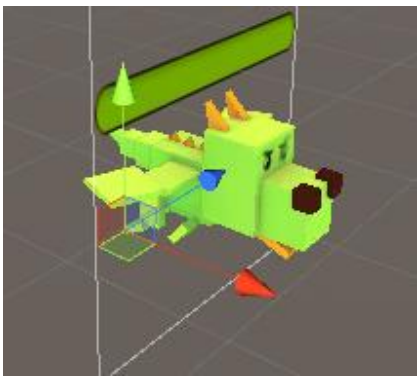

Figura 34. Enemigo dragón

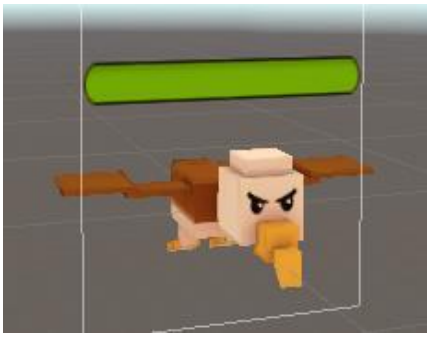

Figura 35. Enemigo cóndor

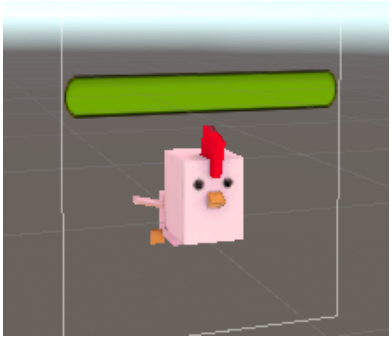

Figura 36. Enemigo gallina

El diseño de estos enemigos ha sido descargado gratuitamente desde la *Asset Store*. Este asset se llama *Free Low Polygon Animal,* el cual incluye las texturas, materiales y animaciones para estos enemigos. A partir de los *GameObjects* que el asset proporciona, se le ha ido añadiendo componentes para definir el comportamiento de estos enemigos. En la Figura 37 se muestra los componentes que tienen todos los enemigos.

Figura 37. Componentes de los enemigos

Los componentes, son un elemento esencial e indispensable en la creación de cualquier juego con Unity3D. Por medio de los componentes, Unity3D aporta funcionalidades ya predefinidas que acortan considerablemente el tiempo para el desarrollo del juego.

En la Figura 37 se muestra los componentes que posee el *GameObject Chicken*, que pertenece a unos de los tres tipos de enemigos presentados anteriormente. A continuación, se definirán la utilidad de cada componente, exceptuando aquellos componentes que tengan relación con la API de *Networking* de Unity3D, que se explicarán más adelante.

- ***Transform:*** Este componente se encarga de definir la posición, rotación y tamaño del *GameObject* al que está asociado este componente. Este componente también aparecía en el componente *Terrain*, y como ya se ha había mencionado anteriormente, aparecerá en todos los *GameObjects* del juego.

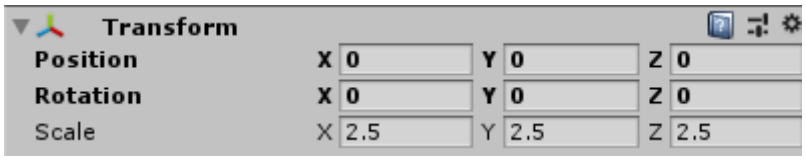

Figura 38. Componente Transform

- ***Animator:*** Especifica qué animaciones realizará este *GameObject*. Las animaciones venían incluidas en el asset *Free Low Polygon Animal.* Son animaciones preprogramadas en base a puntos y su ejecución se desarrolla mediante la activación de interpolación entre esos puntos. El único cambio que se ha hecho ha sido especificar el *Update Mode* a *Normal* y el *Culling Mode a Cull Update Transforms*. De esta forma, la animación de movimiento de los enemigos se ejecutará cada vez que estos se muevan.

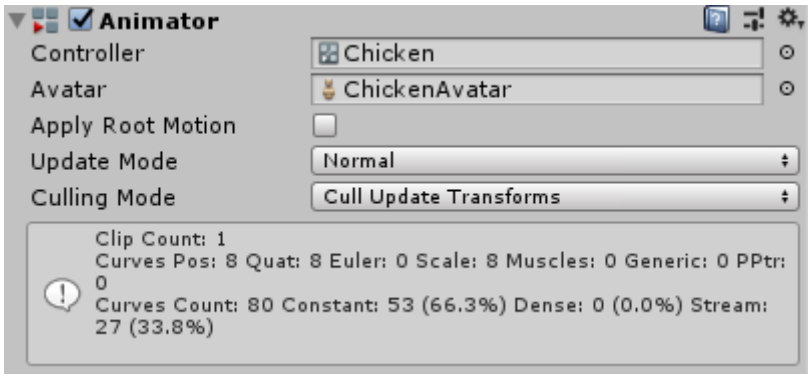

Figura 39. Componente Animator

- ***Box Collider:*** Dota a los *GameObjects* de la capacidad de detectar las colisiones y de que estas sean detectadas por los demás *GameObjects*. Sin este componente el enemigo podría atravesar paredes y no podría recibir daño alguno, ya que la bala le atravesaría el cuerpo como si no existiese.

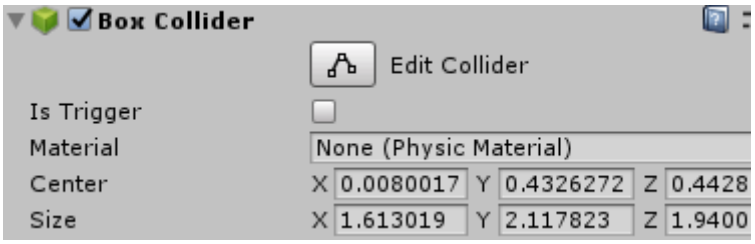

Figura 40. Componente Box Collider

- ***RigidBody:*** Puede dotar al *GameObject* de los parámetros propios de las leyes de la física, entre los que se encuentran; masa, gravedad, arrastre…etc. Además, este componente es necesario que esté presenté en algunos de los *GameObjects* que colisionan para que sea perceptible por el motor físico. Por lo tanto, si dos *GameObjects* colisionan y ninguno de los dos poseen el componente RigidBody, la colisión no se detectará correctamente.
  Otra opción muy importante de este componente es la de *Constraints.* Mediante esta opción se puede limitar el movimiento del *GameObject* en función de los ejes que se marquen como prohibidos.

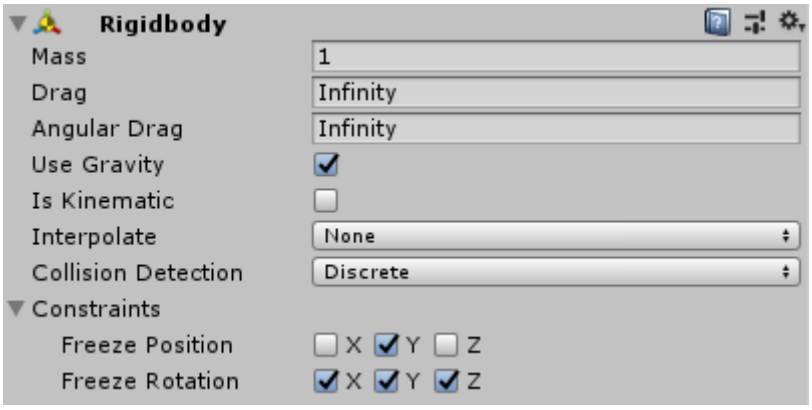

Figura 41. Componente RigidBody

- ***Health:*** Este componente es un script programado con *C#* que ha sido desarrollado para dotar al enemigo de una barra de salud que sea susceptible a los daños que reciba de los enemigos y, además, se sincronice a través de la red para que un enemigo concreto aparezca con la misma cantidad de salud para el resto de jugadores. Este componente ha sido reutilizado tanto para enemigos como para

jugadores. En la Figura 42 se muestran los parámetros de entrada que tiene este componente y cuál es la utilidad de cada uno de ellos.

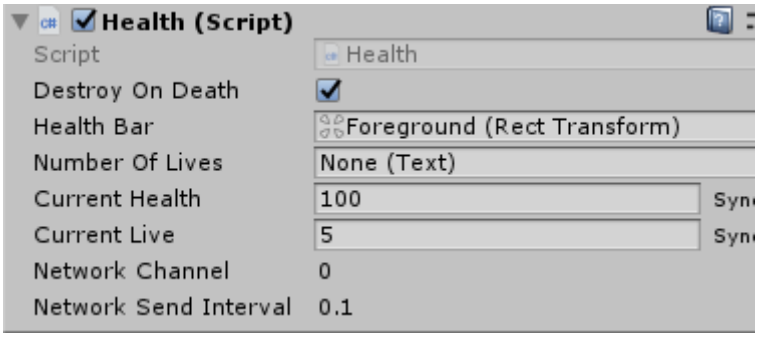

Figura 42. Componente Health en los enemigos

- ***Destroy On Death:*** esta opción es un *booleano* que marca la diferencia entre un jugador y un enemigo. Si la opción *Destroy On Death* está marcada, significa que el *GameObject* será destruido cuando este pierda toda su salud. En el caso de no estar marcado, el *GameObject* no se destruirá, simplemente reaparecerá en otro lugar del mapa mediante la función *RpcRespawn* creada dentro del código.
- ***Health Bar:*** Este parámetro de entrada representa un *GameObject* del tipo *RectTransform.* Este *GameObject* es el elemento gráfico que hará de barra de salud encima de la cabeza del enemigo. Cuando la vida del enemigo haya cambiado, el elemento *RectTransform* variará al nuevo valor para que pueda ser visto por los demás jugadores.
- ***Number of Lives:*** En el caso de un enemigo este valor no tiene utilidad, y por ello no se encuentra asociado a ningún *GameObject* del tipo texto. En el caso de un jugador mediante este parámetro se podrá ver el número de vidas de las que dispone.
- ***Current Health:*** Establece la salud máxima con la que nace el enemigo. Para todos los enemigos y jugadores este valor será 100.
- ***Current Live:*** No tiene utilidad para el enemigo, ya que en el momento que este llegue a 0 de su salud, el *GameObject* se

destruirá. Por lo tanto, el valor 5 de la imagen no tiene ningún impacto funcional sobre los enemigos.

- ***Enemy Partisan***: Este script ha sido desarrollado para que el enemigo detecte la ubicación de un jugador cercano y lo persiga para quitarle vida. El funcionamiento de este script se encuentra comentado paso a paso en el proyecto. Este componente no tiene parámetros de entrada.
- ***Colision:*** Mediante este script el enemigo podrá restar vida al jugador cuando ambos colisionen.

4.5 Jugadores

Son los avatares asociados y controlados por los usuarios que participen en una partida. Inicialmente fueron representados como cápsulas, pero posteriormente se decidió mejorar su diseño descargando un modelo gratuito en la Asset Store para que tenga similitud a la de un androide.

Figura 43. Avatar del jugador

Este *GameObject* tiene multitud de componentes, entre los que se destaca la funcionalidad programada asociada con el streaming de audio y video, gestión de la cámara, control de puntos para la victoria, y el guardado de registros sobre las personas que están transmitiendo datos, etc.

A continuación, se muestra los componentes asociados al jugador y se describirá brevemente su papel. Algunos *scripts* que se han asociado al jugador tienen tareas como la de establecer el *streaming* o comunicar

variables con la API de Networking de Unity3D. Estos *scripts* serán explicados con más detenimiento en las secciones del apartado 5.

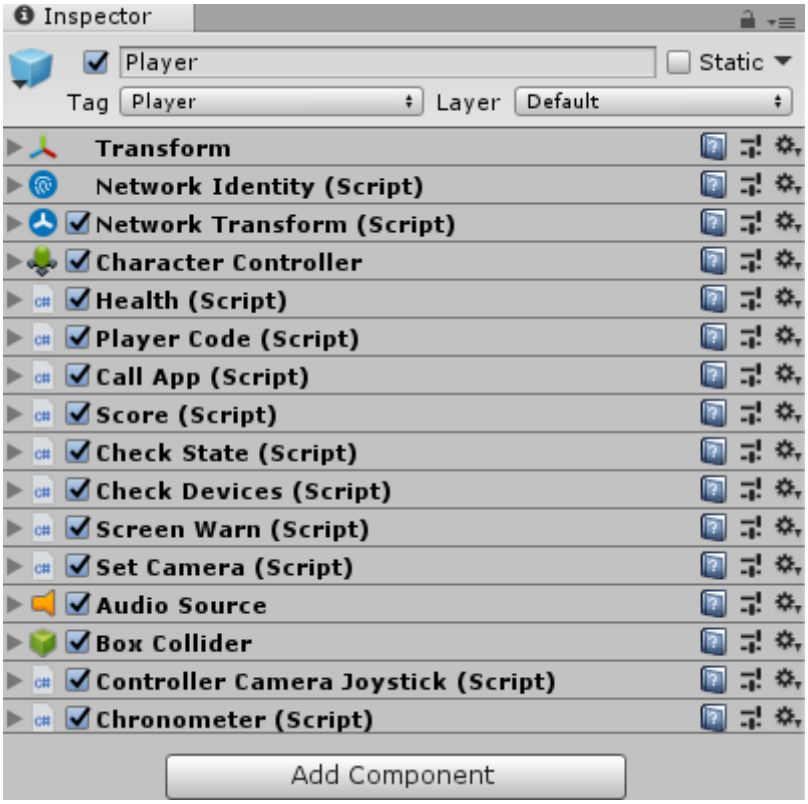

Figura 44. Componentes del jugador

En la Figura 44 aparecen todos los componentes asociados al jugador, pero solo los nuevos no descritos serán descritos a continuación.

- ***Character Controller:*** Permite controlar el movimiento del avatar. Mediante ese componente se puede gestionar momentos como comprobar si el jugador se encuentra colisionando con el suelo, o mover el avatar a una velocidad determinada.

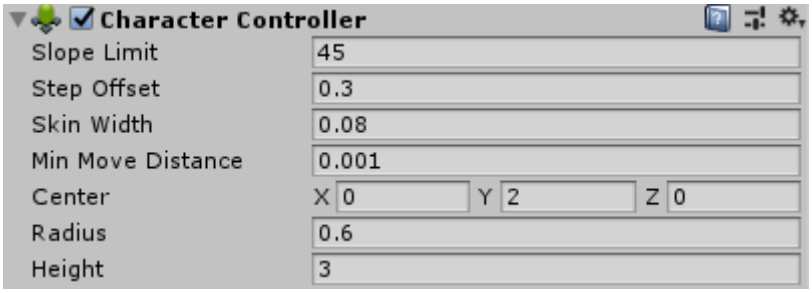

Figura 45. Componente Character Controller

- ***Health:*** Es el mismo componente que tienen los enemigos, y que se describió anteriormente. En este caso, el parámetro de entrada *Number of Lives* no está vacío, ya que en el caso del jugador se le incluye el *GameObject* de tipo texto que aparece en su interfaz de usuario (UI) para que aparezca representado el número de vidas del mismo. El número de vidas con la que empiezan todos los jugadores es de 5, valor que ha sido especificado en el parámetro *Current Live.*

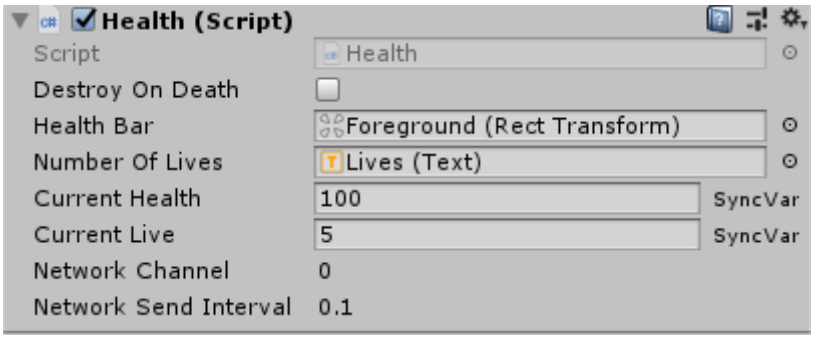

Figura 46. Componente Health en el jugador

- ***Player Code:*** Es el *script* más largo y en el que se ha dedicado gran parte del tiempo a la hora de realizar este proyecto. Realiza numerosas funciones entras las cuales se encuentran:
    - Gestionar el movimiento del jugador por medio del teclado del PC o por medio del joystick derecho para el caso de Android.
    - Gestionar el movimiento de la cámara con el ratón del PC o por medio del joystick izquierdo en el caso de Android.
    - Definir la función de disparo.
    - Discernir en qué momento deberán o no aparecer las pantallas de fin del juego.
    - Diferenciar la UI del jugador entre la versión Android y la versión de PC. La versión de Android trae consigo numerosos botones para hacer más intuitiva y fácil la experiencia de juego.
    - Cargar la lista de *streaming* abiertos y en espera para que un jugador pueda acceder a ellos.
    - Abrir o cerrar el chat del juego, además de permitir escribir texto y enviarlo.
    - Extraer el nombre y color elegido por el usuario en el *Lobby* para que pueda ser mostrado correctamente cuando el juego inicie.

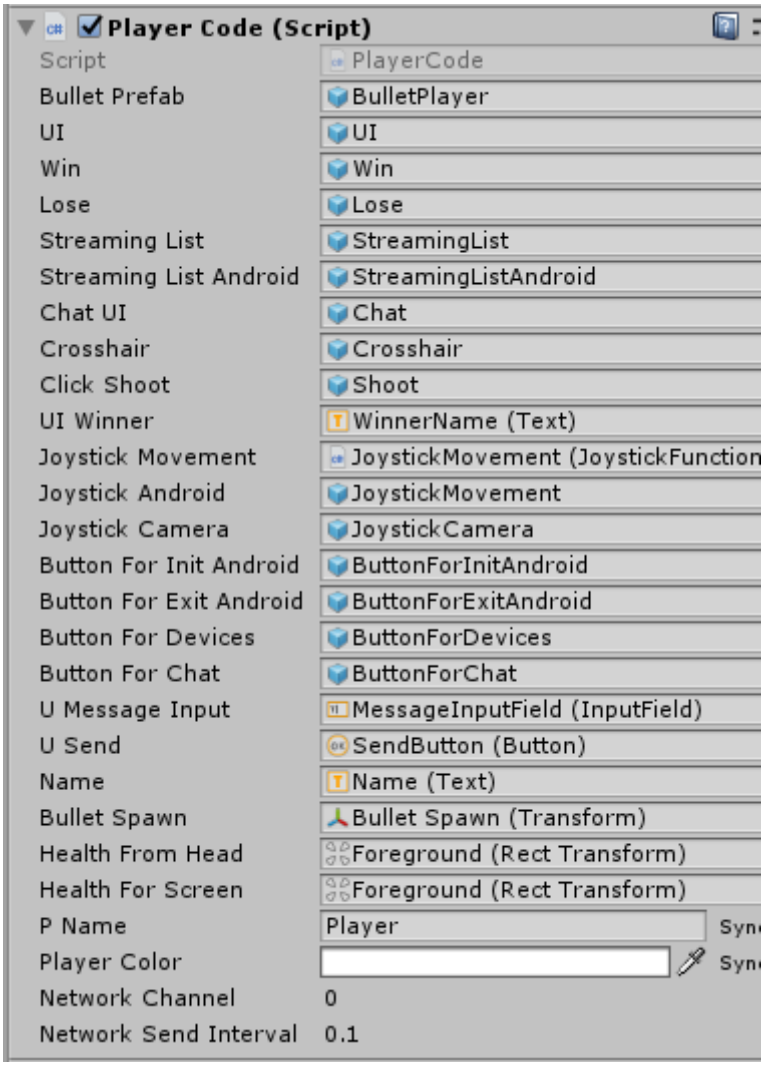

Figura 47. Componente Player Code

Este *script* tiene numerosos parámetros de entrada. Entre los cuáles se encuentran los *GameObjects* de entrada para modificar la interfaz de usuario del jugador cuando se realiza una determinada acción. A continuación, se enumeran únicamente *GameObjects* que sirven para mostrar o esconder mensajes o menús.

- ***UI***: Es la interfaz de usuario donde se encuentra la vida, la barra de salud, los puntos, el *crosshair*…etc.
- ***Win***: Es el mensaje de victoria que le aparecerá al jugador que gane la partida.
- Lose: Es el mensaje de derrota que le aparecerá al jugador o jugadores que pierdan la partida.

- ***Streaming List:*** Muestra la lista de salas activas de streaming para que los jugadores puedan entrar a ellas.
- ***Streaming List Android:*** Es la misma sala que la de Streaming List, pero en este caso es para Android, ya que cambia la posición.
- ***Chat UI:*** Permite mostrar o esconder la ventana del chat.
- Crosshair: Esta es la mira que aparece para apuntar con el arma. Cuando el jugador pierde o gana la partida, desaparece esta mira, para evitar la sobrecarga de elementos en la pantalla.
- ***Click Shoot:*** Es un punto negro que aparece en la pantalla cuando se juega en la versión Android. Cuando este punto es presionado, el jugador disparará con su arma.
- ***UI Winner:*** Muestra el nombre del jugador que ha ganado la partida.
- ***Joystick Android:*** Muestra el joystick derecho en la versión Android que permite al jugador mover el avatar.
- ***Joystick Camera:*** Muestra el joystick izquierdo en la versión Android que permita al jugador mover la cámara del avatar.
- ***Button For Init Android:*** Es un botón que aparece en la versión Android para poder iniciar una sala de streaming. En la versión de PC esta función se realiza con la tecla F1.
- ***Button For Exit Android:*** Es un botón que aparece en la versión Android para poder salir de una sala de streaming. En la versión de PC esta función se realiza con la tecla F10.
- ***Button For Devices:*** Es un botón que aparece en la versión Android para poder encender el micrófono y la cámara cuando el jugador se encuentra dentro de una sala de streaming. En la versión de PC esta función se realiza con la tecla F9.
- ***Button For Chat:*** Permite abrir el chat mediante un botón en la versión de Android. En el PC se abre con la tecla control de la derecha.

Además, hay otro tipo de parámetros que realizan funciones más específicas y que se enumeran a continuación:

- ***Bullet Prefab:*** Se debe incluir aquí el *GameObject* correspondiente a las balas. Mediante la función *CmdFire* del código, el servidor usará este *GameObject* de bala para instanciar las balas correspondientes a los disparos de cada jugador.

- ***Joystick Movement:*** Es un *script* que se introduce como parámetro de entrada para poder usar sus funciones. De esta manera se podrá usar las funciones correspondientes al joystick dentro del código.
- ***Message Input Field:*** Es el campo de entrada de texto del chat donde se escribirá el mensaje que se desea enviar.
- ***Send Button:*** Es el botón del chat que permitirá enviar el texto escrito el campo de texto anterior a través de la red.
- ***Name***: Es un parámetro de entrada de tipo texto en el que aparecerá el nombre que el jugador haya puesto antes de iniciar la partida. Este nombre aparecerá encima del avatar de cada jugador.
- ***Bullet Spawn:*** Es la posición desde la cual deben aparecer todas las balas en el momento en el que estas son disparadas. Esta posición corresponde a la punta de la pistola de cada jugador.
- ***Health From Head:*** Es la barra de salud de color verde que aparece encima de la cabeza del avatar dentro del juego.
- ***Health For Screen:*** Es la barra de salud de color verde que aparece encima de la UI de cada jugador para que este pueda ver su propia vida.
- ***P Name:*** Permite sincronizar el valor de nombre introducido en la escena anterior en donde el jugador debe introducir su nombre.
- ***Player Color:*** Permite sincronizar el valor del color escogido en la escena anterior en donde el jugador debe elegirlo.

4.6 Main Camera

La cámara principal supone un elemento primordial del juego, ya que se encarga de gestionar la visión que tendrá el jugador para controlar su avatar. El juego ha sido desarrollado para jugarlo en primera persona, ya que es el tipo de cámara más extendido para este tipo de juegos de disparos. Además, el juego dispone de una versión para las *Oculus Go*, por lo que el uso de una cámara en primera persona aumenta considerablemente la inmersión que se tiene del juego.

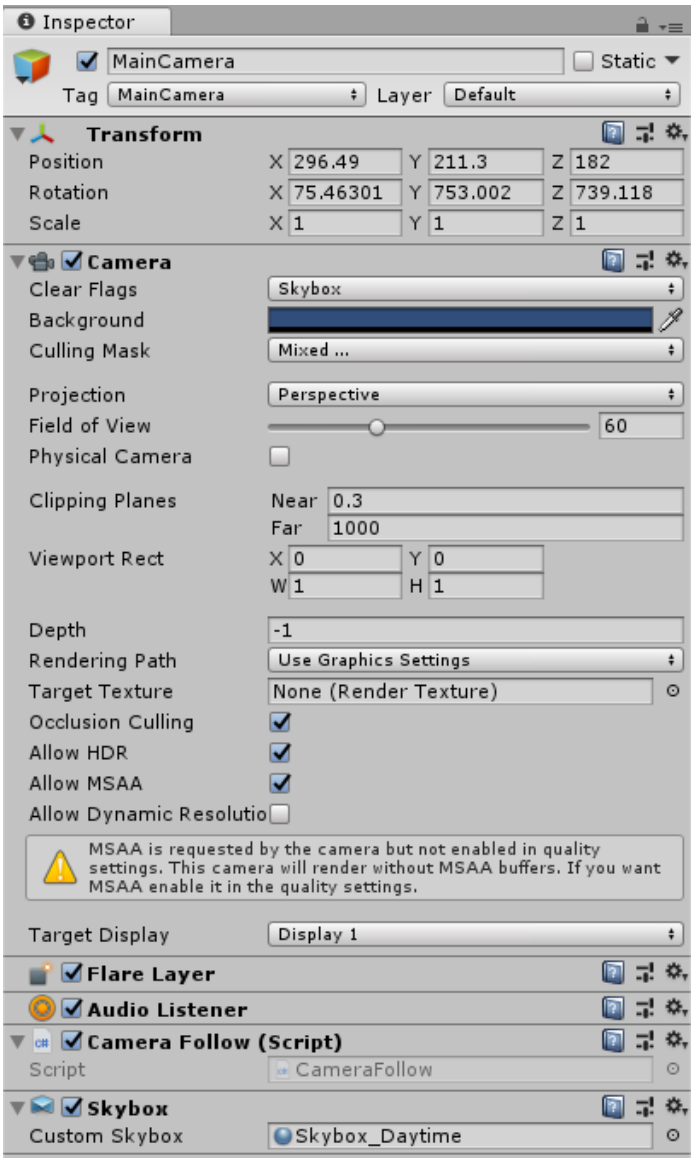

Figura 48.GameObject MainCamera

El componente *Camera* es el que hay que añadir para que el *GameObject* se comporte como una cámara. Las opciones que aparecen escogidas dentro del componente *Camera* son las que trae por defecto. El *GameObject MainCamera* cuenta con tres componentes añadidos manualmente los cuáles son importantes detallar:

- ***CameraFollow:*** Es un *script* que se ha desarrollado para que la cámara siga en todo momento al jugador correspondiente a cada máquina. Para realizar esta tarea, dentro del código se debe especificar la posición que la cámara debe seguir en todo momento dentro de la función *LateUpdate*. Esta función es la especificada por Unity3D para que la cámara siga al jugador y se ejecuta en cada fotograma.

Ya que anteriormente se ha dicho que el juego se juega en primera persona, la cámara debe posicionarse delante del avatar.

- ***Audio Listener:*** Permite percibir sonidos dentro juego, solo puede haber un componente de este tipo en cada escena.
- ***Skybox:*** Mediante este componente, se puede elegir el cielo que aparecerá dentro del juego. El cielo que ha sido escogido se ha descargado de la Asset Store y el nombre de este paquete se llama *Free HDR Sky.*

Este tipo de configuración de la cámara sirve sólo para jugar en la versión de PC o Android. En la versión de las *Oculus Go* la cámara tendrá varios *scripts* que serán obligatorios emplear para que funcione con los mandos de la *Oculus*. En este capítulo solo se muestra el desarrollo del juego en su versión de PC o Android, en un capítulo posterior se mostrará cómo se ha desarrollado el juego para la versión de las *Oculus Go*.

4.7 Game Manager

Es un *GameObject* que se ha creado para estar presente dentro de la escena del juego. Este *GameObject* contiene un *script* asociado con la inteligencia del juego y encargado de la ejecución de las reglas del mismo, este *script* se llama *Game Control.* Si este *GameObject* no se pusiera en la escena del juego, no habría control sobre quién gana o pierde la partida. Por lo tanto, la partida duraría infinitamente.

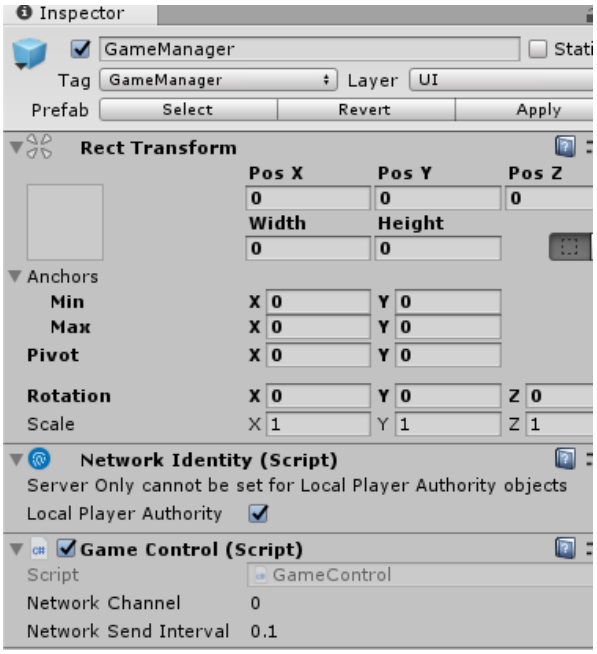

Figura 49. GameObject Game Manager

El *script Game Control* realiza las siguientes funciones de control dentro del juego:

- Se encarga de comprobar las vidas restantes de todos los jugadores de la partida. En el momento en el que todos los jugadores exceptuando uno pierdan todas las vidas, mandará un aviso para indicar el final de la partida y aparecerá el mensaje de victoria o derrota correspondiente a cada jugador.
- Indicará que la partida ha finalizado cuando el temporizador global de la partida llegue a 0.

4.8 Call Manager

Este *GameObject* ha sido creado para que realice un seguimiento de los jugadores que tienen una sala de streaming iniciada, aquellos que son capaces de entrar a dicha sala y aquellos que se encuentran ya dentro de una sala. Mediante este *GameObject* se controlará el color que aparece en la parte superior de cada jugador. Cada color simboliza un estado diferente en relación a la funcionalidad de *streaming* del juego. Existen tres colores posibles para cada estado:

- ***Punto verde:*** Cuando un jugador tiene un punto de color verde al lado de su barra de vida, esto indica que el jugador ha iniciado una sala de *streaming*, y está esperando que otro jugador entre a ella.
- ***Punto rojo:*** Simboliza que el jugador se encuentra comunicándose con otro jugador en alguna sala de *streaming*. Por lo tanto, si quiere conectarse a otro jugador, deberá primero salirse de la sala en la que está.
- ***Punto amarillo:*** Simboliza que el jugador no ha creado ni ha entrado a ninguna sala de streaming. Por lo tanto, está desocupado.

Es importante tener en cuenta, que las salas de streaming son punto a punto, por lo tanto, solo podrán estar como máximo dos jugadores en una sala de *streaming*. Si un jugador que está dentro de una sala de streaming quiere comunicarse con otro que se encuentre fuera de esta, deberá primero salirse de su sala inicial y unirse a la del jugador con el que quiere comunicarse, siempre y cuando alguno de los dos jugadores tenga una sala abierta.

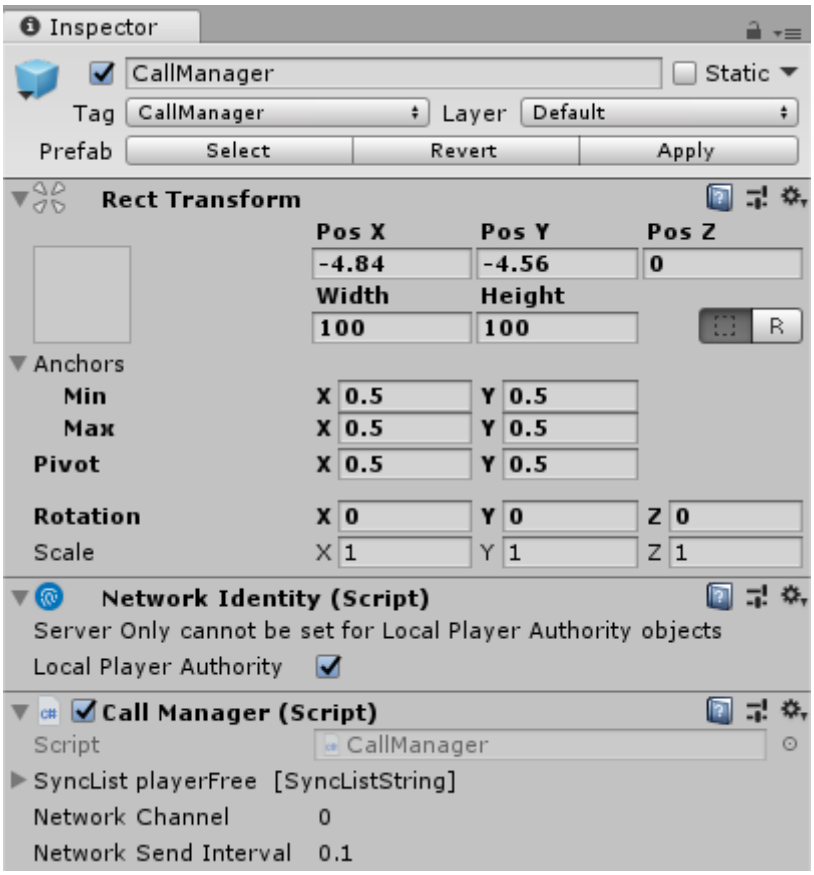

Figura 50. GameObject Call Manager

5. Funciones de red en Unity3D

En el capítulo 4se describe cómo se ha desarrollado el juego incidiendo más en las herramientas que proporciona Unity3D para crear un juego de un solo jugador. En este capítulo se mostrará en primer lugar cómo Unity3D ofrece la posibilidad de dotar a sus juegos con funciones multijugador, junto con una explicación de cómo se han aplicado estas funcionalidades. Finalmente, se hará lo mismo para describir cómo se ha realizado la incorporación de las funciones de *streaming* al juego.

5.1 API Networking de Unity3D

La API de alto nivel multijugador de Unity3D (HLAPI) es un conjunto de funciones desarrolladas para incorporar capacidades multijugador en los juegos desarrollados con Unity3D. Estas funciones nos ayudan a incorporar las tareas que cualquier juego multijugador debe poseer, sin preocuparnos del funcionamiento en los niveles inferiores ya que la HLAPI está construido sobre la capa de transporte y aporta una interfaz sencilla para poder tratar con ella.

HLAPI funciona con la arquitectura cliente-servidor. El servidor también puede comportarse como un cliente, por lo que puede ser un jugador más de la partida a efectos prácticos. Por lo tanto, si en mitad de la partida el jugador que hace también de servidor se desconecta, los demás clientes también se saldrían de la partida, puesto que se quedarían sin servidor.

Este conjunto de funciones se encuentra integrado dentro del *namespace* ***Unity3DEngine.Networking***. De modo, que cualquier *script* en donde se quiera usar las funciones de la API de Networking deberán incluir este *namespace*.

De manera resumida, la HLAPI nos ha permitido realizar las siguientes tareas dentro de este proyecto:

- Implementar un sistema de espera para iniciar partidas simultáneamente, esto se consigue mediante el componente ***Network Lobby Manager***, el cual se describe en la sección 5.2.1.2. Este componente permite además limitar el número de jugadores y el número de conexiones que podrán participar en una misma partida.
- La sincronización de la posición y el movimiento de todos los jugadores y enemigos. Eso se consigue mediante el componente *Network*

Transform. Además, permite la posibilidad de instanciar elementos en todas las escenas mediante la clase *Network Scene*.

- Utilizar *Remote Procedure Calls* o llamadas a procedimientos remotos entre el servidor y el cliente, o viceversa. Mediante este sistema se ha podido utilizar variables sincronizadas entre el cliente y servidor para gestionar el sistema de puntos, el contador de vidas, la disponibilidad del *streaming*, entre otros aspectos que se describen en la sección 5.2.
- Gestionar parámetros de las capas inferiores para conseguir la conectividad entre los equipos. Además de poder configurar el tipo de calidad de servicio utilizado para el envío de información.
- Gestionar el control de salud de los enemigos y los jugadores.

Según el manual, HLAPI cuenta con esta serie de capas que aportan funcionalidades a sus proyectos, de las cuáles sólo se emplearán algunas de ellas. La siguiente Figura 51 muestra la tabla de funcionalidades que nos proporciona el manual de Unity3D.

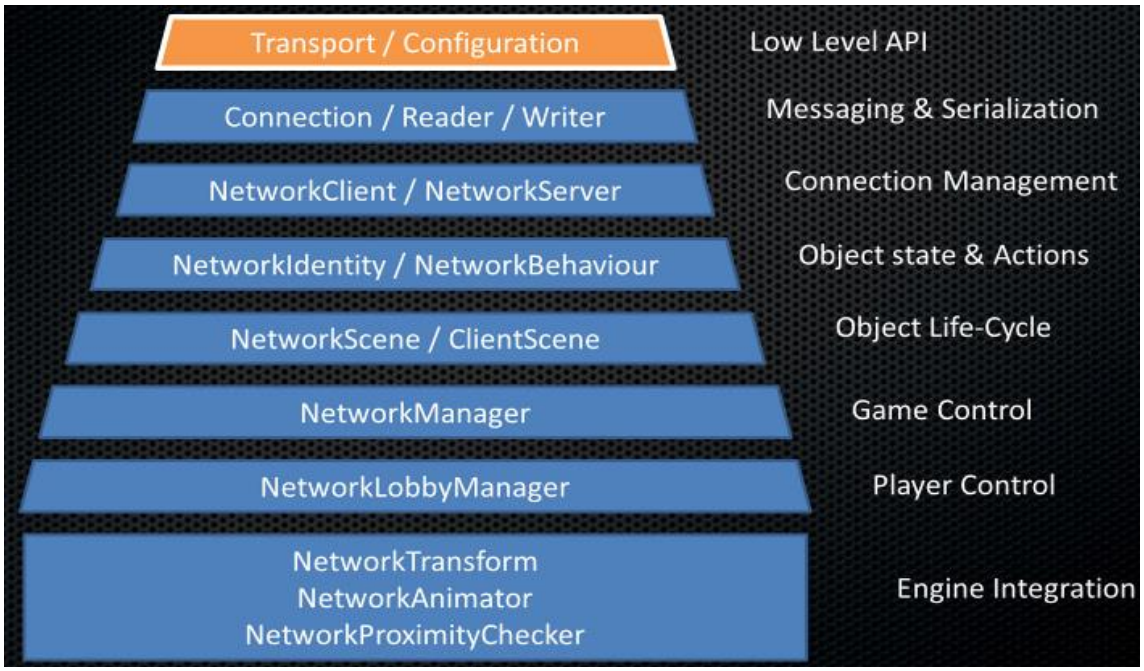

Figura 51. Funcionalidades generales de HLAPI

5.1.1 Arquitectura de HLAPI

Antes de entrar en profundidad en las funcionalidades proporcionadas por HLAPI y en cómo implementarlas dentro del juego, es indispensable entender cómo funciona su arquitectura básica. HLAPI sigue una arquitectura servidor-cliente:

- **Servidor:** Es la instancia del juego a la que los demás clientes se conectan cuando quieren jugar en un entorno multijugador. La instancia del servidor es la encargada de gestionar la aparición/desaparición de los *GameObjects* del juego, y cuya escena será de la que deriven los demás clientes. El servidor puede funcionar de dos modos diferentes:

    - **Servidor dedicado:** Es una instancia que única y exclusivamente se comporta como servidor, es decir, sólo aparece la escena del juego funcionando.

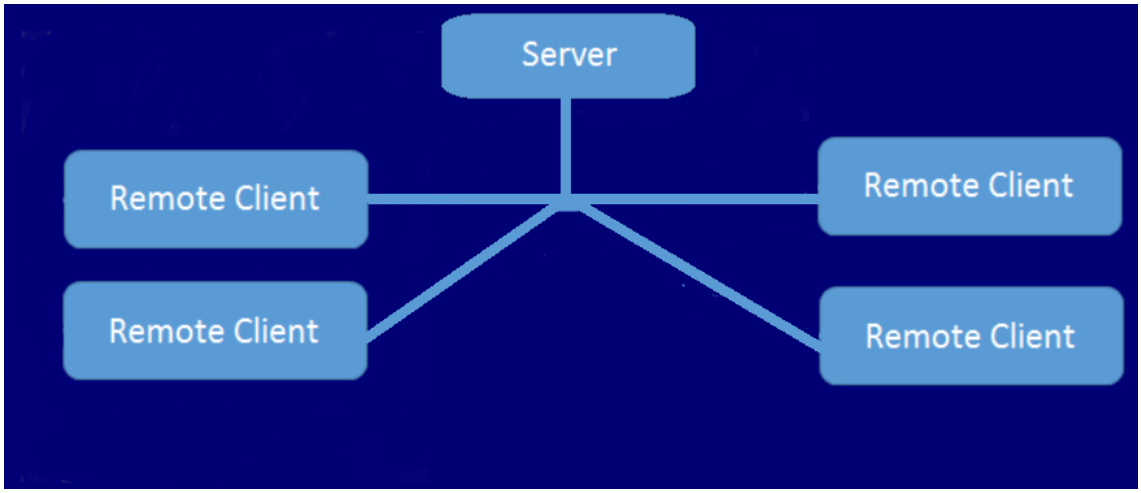

Figura 52. Arquitectura con "servidor dedicado"

En la Figura 52 se puede ver como los clientes remotos acceden se conectan al servidor a través de su dirección IP.

    - **Servidor de *host*:** Es aquel en el que además de comportarse como servidor de la partida, es un jugador más de ella. Este jugador no tendrá ningún tipo de ventaja sobre los demás, únicamente jugará la partida como cualquier, pero internamente estará enviando datos constantemente por medio de la API de Networking ya que la máquina se comporta también como un servidor.

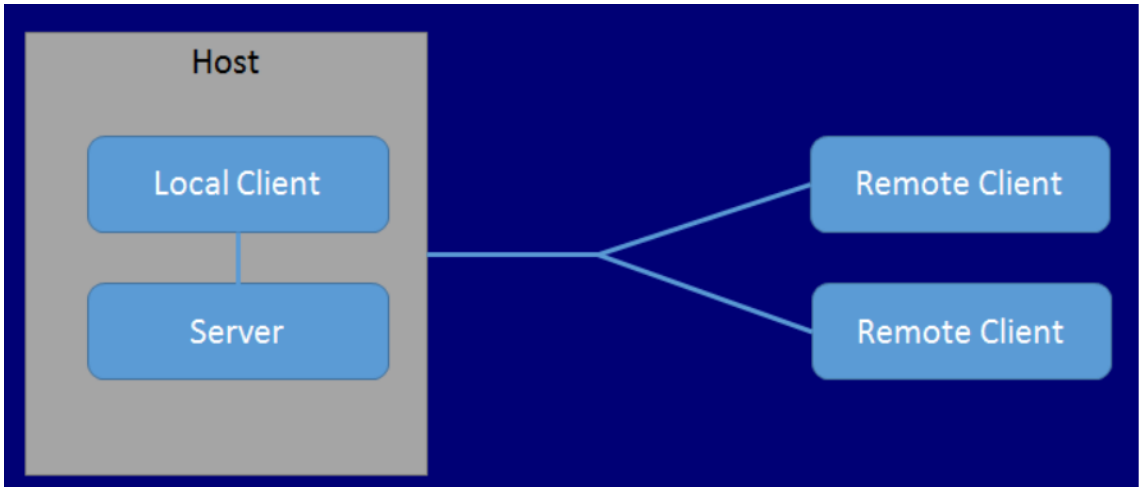

Figura 53. Arquitectura con "servidor de host"

Se puede ver en la Figura 53 como tenemos un equipo representado de color gris, dentro de él hay dos instancias: una de un cliente normal y la otra de un servidor. A efectos prácticos, el cliente local no deja de ser un jugador como otro cualquiera, aunque si a este cliente le falla la conexión, la partida se terminaría para todos, ya que es él el que ejecuta la instancia servidora. Para solucionar este problema, se optaría por emplear un servidor dedicado, de esta manera ningún jugador tendría el peso de la partida en él. Es muy importante saber qué tipo de servidor se va a querer utilizar para el juego, ya que puede variar un poco la programación.

- **Cliente:** Son instancias del juego que se conectan al servidor, para ello estas deben encontrarse dentro de una misma red o reconocer la dirección IP del servidor. Además, se debe de especificar un puerto. Con esta información, Unity3D se encarga de conectarlos a bajo nivel, sin necesidad de que el desarrollador se preocupe de algo más a dicho nivel.

5.1.2 Componentes para implementar la API en Unity3D

La manera que tiene Unity3D de implementar funcionalidades a los *GameObjects* es mediante los componentes, tal y cómo se ha ido viendo a lo largo de este documento. Esto no cambia para el caso de la API de *Networking*, la cual también se sirve mediante componentes ya desarrollados por Unity3D. Este capítulo se subdividirá en varios apartados en donde se presentarán y explicarán los componentes que han sido utilizados en este juego.

5.1.2.1 Componente Network Lobby Manager

Es el componente central de la API, mediante él se deben especificar numerosos parámetros importantes para realizar la conexión entre equipos. Para que el juego esté lo mejor hecho posible, se ha decidido buscar un *asset* que proporcione facilidades para crear las salas de configuración o *Lobby*. El *asset* se llama ***Network Lobby***, y ofrece menús y componentes que han sido ligeramente modificados para adaptarlos a las necesidades de la creación de salas. A continuación, en la Figura 54 se muestra el componente ***Lobby Manager***, el cuál es una versión adaptada (por el asset descargado) del componente *Network Lobby Manager*.

Este componente básicamente realiza las mismas funciones que el componente *Network Lobby Manager* de Unity3D, ya que muchas de las funciones que utiliza heredan de este. La única diferencia es que este nuevo componente traído por el *asset* incluye parámetros más específicos para la creación de una sala de partidas propias de cualquier juego actual en el mercado, por ejemplo; el tiempo de espera desde que todos los jugadores están listos hasta que inicia la partida. A continuación, se mostrará en la siguiente Figura 54 el componente que trae el asset *Network Lobby*, posteriormente se enumerarán las partes importantes para comprender el funcionamiento de este componente.

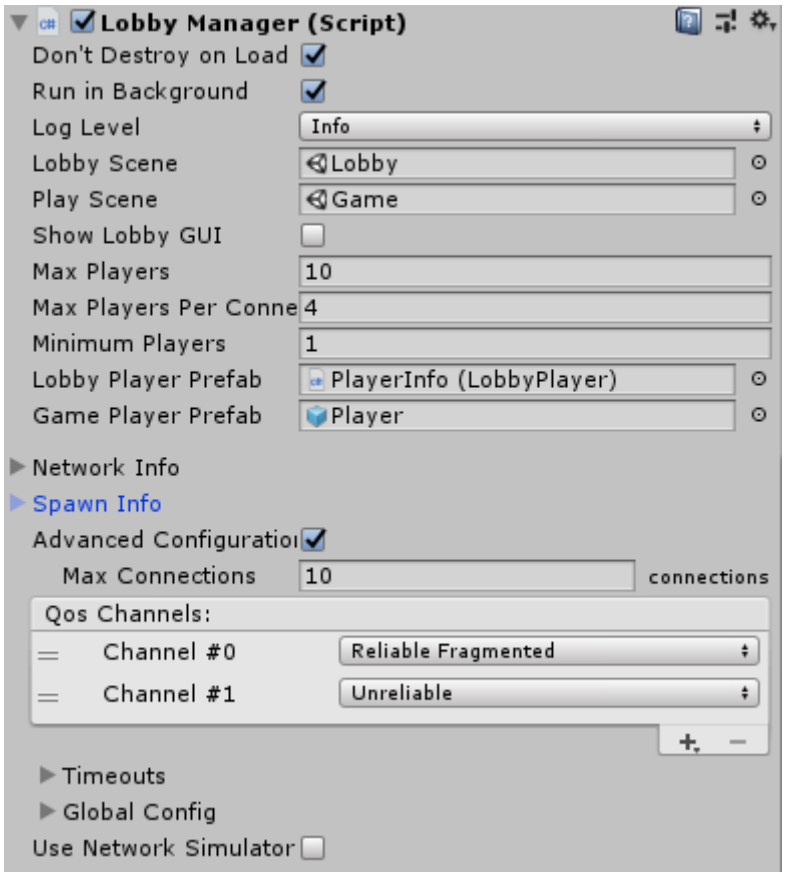

Figura 54. Componente Lobby Manager

- ***Lobby Scene:*** Se especifica la escena inicial que tendrá la función de servir como salas para los jugadores. En este caso se trata de la escena llamada *Lobby*, la cual dispondrá de un menú inicial donde el usuario podrá elegir el tipo de conexión que desea realizar. Este menú viene incluido dentro del *asset* de *Network Lobby*, aunque se ha tenido que realizar algunas modificaciones como el cambio de idioma o la colocación del nombre del juego en la parte superior del menú.

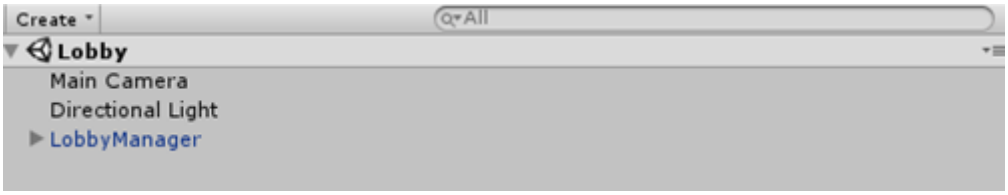

Figura 55. Escena Lobby

- ***Play Scene:*** Se especifica la escena del juego que se ha creado en el apartado 3 de este documento. Una vez que en el *Lobby* el jugador ha elegido su color y su nombre dentro de una partida, y posteriormente

todos los jugadores han marcado la opción Entrar, aparecerá un temporizador de 5 segundos en la pantalla de cada jugador al mismo tiempo. Cuando el contador llegue a 0, el juego en todos los jugadores cambiará a la escena de juego (la escena *Game*). En la Figura 56 se muestra los *GameObjects* de los que está compuesta les escena Game, la cuál ha sido tratada en el capítulo 4.

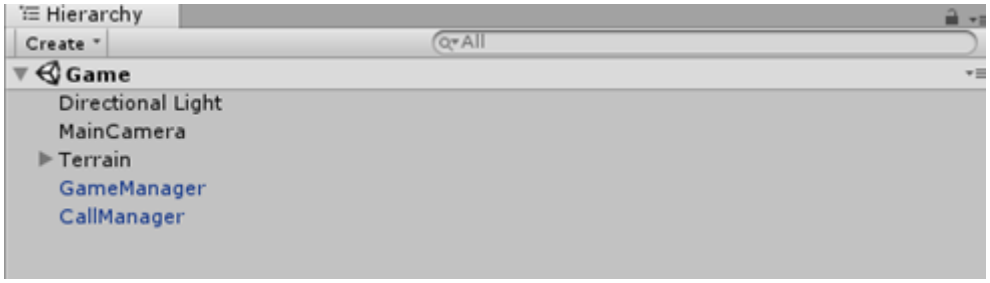

Figura 56. Escena Game

- ***Max Players:*** Es el número de jugadores máximo que podrá tener el juego en una misma partida. Se ha decidido limitar la partida a un número de 6 jugadores como máximo. De esta manera, se intenta mantener una relación entre el número de jugadores y el tamaño del mapa para que no entorpezca la experiencia de juego.
- **Max Players Per Connection:** El número de jugadores que entrar a una partida desde una misma IP, se ha puesto 4 porque ha sido necesario para realizar diversas pruebas.
- ***Game Player Prefab:*** Se especifica el *GameObject* que hará de avatar del jugador dentro de la partida. En el caso de este proyecto se debe poner el *GameObject* del Player.

En la Figura 54 se puede apreciar que hay ventanas que se encuentran no desplegadas dentro del componente Lobby Manager. Estas ventanas contienen más parámetros y funcionalidades importantes. La Figura 57 pertenece a la ventana desplegaba llamada ***Network Info***:

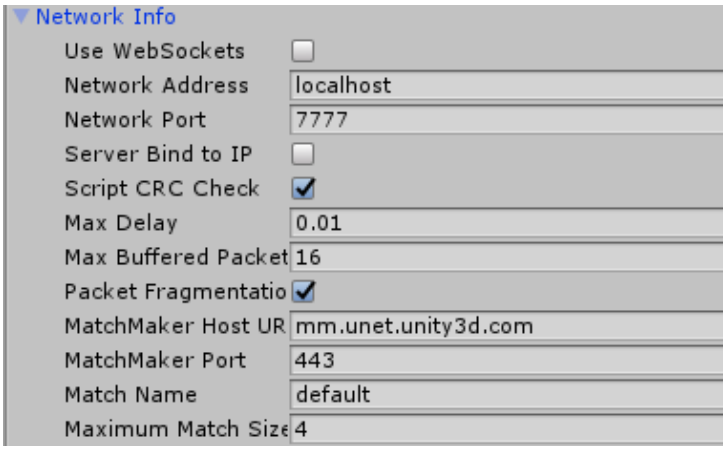

Figura 57. Desplegable Network Info

En la Figura 57 aparecen parámetros de gran importancia para el nivel de transporte. A continuación, se enumerarán algunos de los cuáles es necesario tener en consideración para que el juego funcione.

- ***Use WebSockets:*** Esta opción se encuentra desmarcada ya que no se utilizará la compilación con WebGL que proporciona Unity3D.
- ***Network Address:*** Especifica la dirección IP que hará de servidor en el caso de optar por una conexión manual del tipo servidor dedicado o host.
- ***Network Port:*** El puerto de la conexión, por defecto será 7777.
- ***Server Bind To IP:*** Sirve para que una dirección IP sea la única que pueda conectarse como servidor. Se desmarca esta opción ya que se va a permitir que cualquier jugador pueda comportarse también como servidor.
- ***Script CRC Check:*** Mediante esta opción Unity3D realiza una comprobación entre el servidor y los clientes para asegurarse de que los *scripts* no presenten diferencias. Unity3D recomienda desmarcar esta opción si se va a emplear proyectos diferentes, aunque en este caso está marcada ya que ha sido útil a la hora de desarrollar el juego para ayudar en la depuración.

Los siguientes parámetros; ***Max Delay***, ***Max Buffered Packet*** y ***Packet Fragmentation***, se han mantenido conforme venían por defecto. El parámetro ***MatchMaker Host URL*** y posteriores también venían por defecto. El *MatchMaker Host URL* tendrá la dirección que proporciona Unity3D como su servidor de retransmisión.

Posteriormente se tiene el desplegable ***Spawn Info***. En este desplegable se especificarán aquellos *GameObjects* que serán creados en la escena durante la ejecución del juego. Por lo tanto, aquellos *GameObjects* que vayan a ser creados por el servidor dentro de la escena deberán aparecer en este desplegable para que sus parámetros estén sincronizados en todos los clientes.

Figura 58. Desplegable Spawn Info

En la Figura 58 aparecen aquellos *GameObjects* del juego que ser crearán durante la ejecución de la partida. Aparecen además dos opciones en la parte superior de la lista:

- ***Auto Create Player:*** Permite mostrar un jugador nuevo automáticamente cuando en la escena *Lobby* el usuario entre o cree una nueva partida. Si no se marca esta opción, cuando se cree una partida o se acceda a ella nuestro nombre no aparecerá en la pantalla de la lista de jugadores. Por ejemplo, si se crea una partida desde la opción *Host* y la opción *Auto Create Player* no está marcada, aparecerá lo que se muestra en la Figura 59.

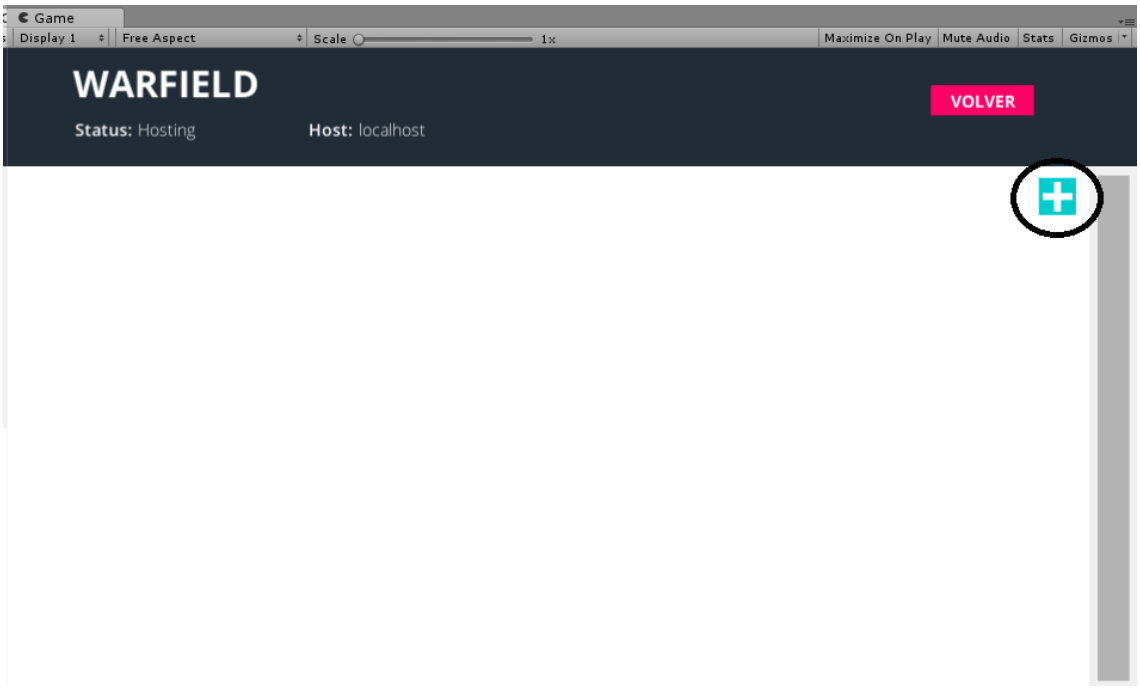

Figura 59. Acceso sin Auto Create Player

Para que aparezca nuestro nombre habrá que hacer click en el "+" que aparece redondeado con un círculo negro en la Figura 59.

- ***Player Spawn Method:*** Esta opción especifica la forma en la que aparecerán los jugadores en el escenario del juego. Se ha marcado la opción ***Random*** para que el juego sea más impredecible, y ningún jugador tenga ventaja sobre otro a la hora de aparecer en el mapa. Hay dos métodos posibles según la forma de aparición:

    - Random: Los jugadores aparecerán aleatoriamente en aquellas zonas del mapa en donde se haya programado que deban salir los jugadores. Para marcar la posición de inicio de los jugadores se hace uso del componente *Spawn Position*.
    - Round Robin: Los jugadores aparecerán secuencialmente entre todos los puntos del mapa disponibles para la aparición de los jugadores.

Para señalar las zonas del mapa como puntos de aparición de los jugadores, habrá que crear un *GameObject* al cual se le ha llamado **Spawn Position** y atribuirle el componente ***Network Start Position***.

5.1.2.2 Componente Network Identity

Este componente lo poseen todos los *GameObjects* que vayan a utilizar funciones que hereden del paquete ***Unity3DEngine.Networking,*** por lo tanto, cualquier elemento del juego que vaya a usar las funciones de la API de *Networking* obligatoriamente deberá llevar el componente **Network Identity** adjunto a él. El componente se encarga de asignar un identificador único al *GameObject* de la escena para que sea reconocible y diferenciable del resto, por todos los demás clientes. A este identificador, el cual es asignado dinámicamente cada vez que un *GameObject* es creado en la escena, se le conoce como **netId**. NetId es un entero de 32 bits, y es único para todos los *GameObjects* instanciados en la escena.

En la Figura 60 se muestra como se ve este componente, y las opciones disponibles para él:

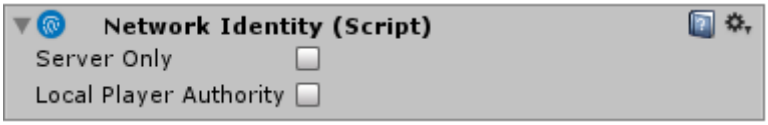

Figura 60. Componente Network Identity

- ***Server Only:*** Se marca esta opción para asegurarse de que el *GameObject* que contiene este componente solo aparezca en el servidor. Por lo tanto, si el servidor instancia este componente en la escena, solo será visto por él mismo, y los demás clientes no lo renderizarán en su escena. Esta opción ha sido marcada en las posiciones de aparición de los enemigos (los GameObjects de Spawn Position) donde no es necesario que el cliente las renderice en su escena. Se debe recordar, que mediante la función Spawn el servidor puede instanciar un *GameObject* para todos los clientes conectados a él.
- ***Local Player Authority:*** Da autorización a los clientes de visualizar este *GameObject*, por lo tanto, es lo opuesto a la opción *Server Only*. Mediante esta opción el *GameObject* instanciado en la escena por el servidor será también visto por los demás clientes. Un ejemplo serían las balas de los jugadores, ya que incorporan esta opción para que cuando sean instanciadas puedan ser vistas por todos los clientes.

Como ya se ha comentado anteriormente, para instanciar un *GameObject* concreto en la escena, el servidor debe emplear la función Spawn. Cuando se ejecuta esta función, Unity3D se encarga de asignar dinámicamente el identificador de red a este *GameObject* nuevo. Para que este *GameObject* sea creado correctamente, debe ser obligatoriamente especificado en la lista de la Figura 58.

5.1.2.3 Componente Network Transform

El componente ***Network Transform*** sincroniza el movimiento y la rotación de los *GameObjects* cuya información se transmite en la red. Este componente solo sincroniza *GameObjects* que han sido creados por desde la red, es decir, aquellos que tienen el componente *Network Identity* adjuntos a ellos. Este componente se ha adjuntado a muchos elementos; como los enemigos, jugadores, balas, manzanas (son *GameObjects* que se reparten a lo largo del mapa para sanar al juego), etc.

En la Figura 61 se muestra este componente añadido a un enemigo, a partir de ella se explicará las partes correspondientes:

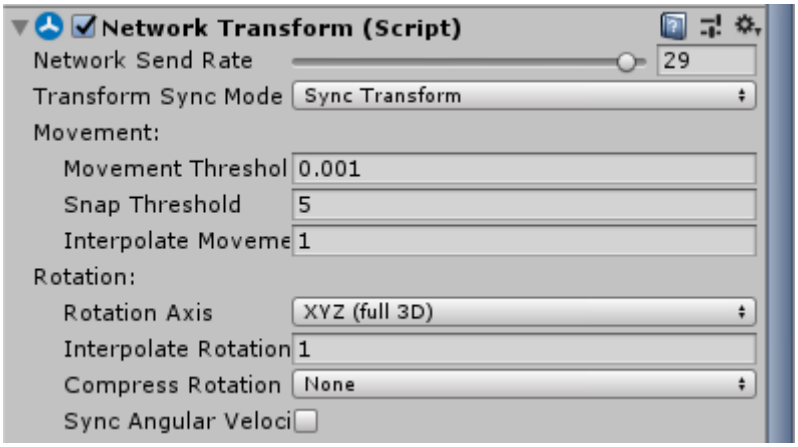

Figura 61. Componente Network Transform

- ***Network Send Rate:*** Establece el número de actualizaciones por segundo a través de la red que recibirá el *GameObject*. Si el valor se pone a 0, no se actualizará la posición del *GameObject* nada más que al momento de este ser creado. En el caso de los enemigos y los jugadores, como se muestra en la Figura 61, el valor es el máximo posible, ya que de esta forma el movimiento de estos será lo más fluido

posible. Sin embargo, en el caso de las balas el valor se ha establecido a 0, ya que, en el código de disparo, antes de ser creada la bala en todos los clientes, se le aplica a esta una fuerza cinética concreta que será la misma tanto para el servidor como para el cliente. Por lo tanto, no se necesita actualizar constantemente la posición. A continuación, se muestra en la Figura 62 el fragmento de código de disparo.

```
693     [Command]
694     public void CmdFire()
695     {
696         // Create the Bullet from the Bullet Prefab
697         var bullet = (GameObject)Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);
698 
699         // Add velocity to the bullet
700         bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward * 150;
701 
702         //Identify bullet
703         bullet.name = pName;
704 
705         // Spawn the bullet on the Clients
706         NetworkServer.Spawn(bullet);
707 
708         // Destroy the bullet after 2 seconds
709         Destroy(bullet, 2.0f);
710     }
```

Figura 62. Código para disparar una bala

Se puede apreciar como en la línea 700, se le aplica una velocidad concreta a la bala, la cuál será la misma tanto para los clientes y el servidor. Por lo tanto, no se necesita actualizar ninguna posición ya que la bala irá a la misma velocidad en todas las instancias del juego y será actualizada la posición por el motor de física.

- ***Transform Sync Mode:*** Especifica a qué componente se le añadirá la sincronización. En la Figura 63 correspondiente al componente ***Network Transform*** de un enemigo, este parámetro tiene el valor ***Sync Transform***, por lo tanto, sincronizará el componente *Transform* del dicho *GameObject* a la red a una cadencia de 29 veces por segundo. Recordemos que el componente *Transform* es el que marcaba la posición, rotación y tamaño del *GameObject* al que está unido.

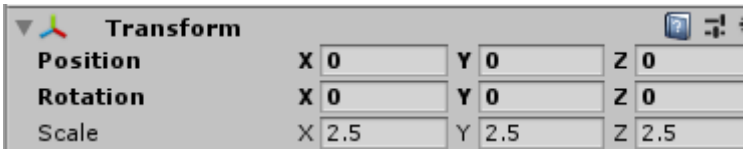

Figura 63. Componente Transform a sincronizar

En el caso de las balas, se debe elegir la opción de ***RigidBody*** en el parámetro ***Transform Sync Mode***. Recordemos que el valor se *Network Send Rate* para las balas es 0, por lo tanto, la única vez que se actualiza el parámetro escogido en *Transform Sync Mode* es justo al principio de que la bala es creada. Ese es el momento en el que bala es creada y se le aplica la fuerza cinética.

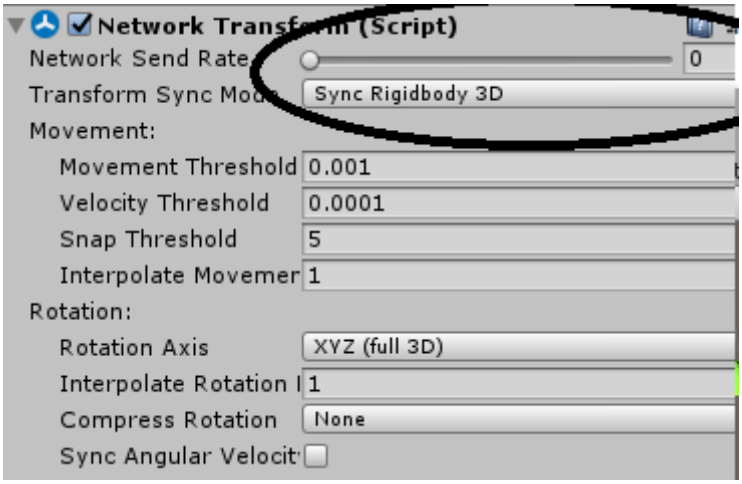

Figura 64. Componente Network Transform del GameObject Bullet

En el caso de los jugadores, hay un componente llamado ***Character Controller***, tiene un comportamiento similar al de *Transform* ya que permite mover el avatar con los controles más fácilmente. Por lo tanto, será este el componente que se atribuya al parámetro *Transform Sync Mode* para ir actualizando el movimiento del jugador en todos los clientes conectados en red.

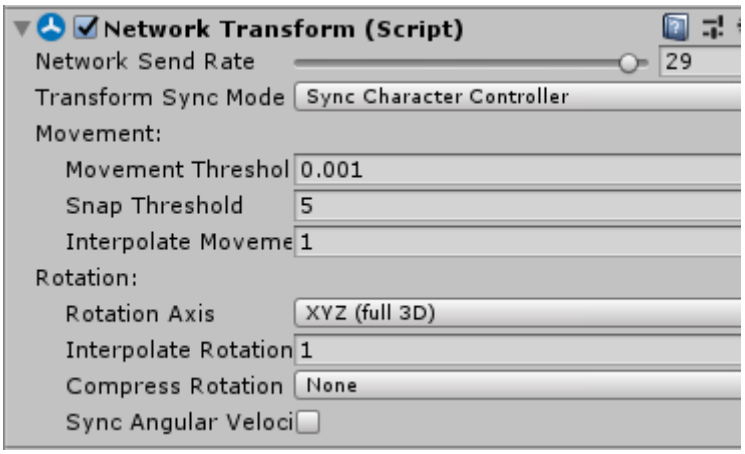

Figura 65. Componente Network Transform del jugador

Para las siguientes variables se ha mantenido los valores por defecto:

- **Movement Threshold:** Establece la distancia en la que un *GameObject* puede moverse sin necesidad de enviar una actualización de la posición de su posición a la red.

- **Snap Threshold**: Establece el umbral en el que, si una actualización de movimiento pone un *GameObject* más lejos de su posición real, el *GameObject* se ajusta a la posición en lugar de moverse suavemente.

- **Interpolate Movement Factor:** Este valor define como de continuo o discontinuo es el movimiento del *GameObject.* Si se reduce este factor el movimiento se verá discontinuo y "a saltos". Por defecto su valor es 1, y será el utilizado en el proyecto.

El siguiente parámetro importante es el de *Rotation Axis*, que especifica en qué ejes se procederá a realizar la actualización del *GameObject*. Al ser un juego en 3D, se especificará que se actualicen los ejes XYZ ya que se querrá actualizar en todo momento a qué dirección mira el enemigo, jugador, etc.

5.1.3 Remote Procedure Calls

Los RPC es el sistema más utilizado en el proyecto para realizar acciones a través de la red, y para modificar variables en todos los clientes casi simultáneamente. Forman parte de la clase ***Network Behaviour***, por lo tanto, el *script* que deba utilizar las RPC deberá heredar de dicha clase. Hay tres tipos de RPC, pero en este proyecto solo han sido necesarias utilizar las dos primeras:

- **Commands**: Son funciones que son llamadas desde el cliente y ejecutadas en el servidor. Deben ser declaradas con una etiqueta **[Command]** en la línea anterior a la función, y el nombre de dicha función debe empezar por **Cmd**. Cuando un cliente llama a esta función, realmente es el servidor el que la ejecuta. La Figura 62 mostrada anteriormente es un ejemplo de función *Command*, pero hay muchas más como la que se muestra ahora en la Figura 66.

```
[Command]
public void CmdNotSending()
{
    isSending = false;
}
```

Figura 66. Función Command

  Mediante estas funciones, el cliente puede modificar parámetros propios y que estos sean perceptibles por el resto de la red. La función de la Figura 66 permite a un jugador modificar un *booleano* así mismo para avisar a los demás jugadores de que ha abierto una sala de *streaming.*

- **ClientRPC:** Son funciones llamadas desde el servidor y que son también ejecutadas en todos los clientes. Es el servidor el que únicamente puede llamar a estas funciones, por lo tanto, si lo hace un cliente aparecerá un aviso en el editor de Unity3D informando de que la acción no ha podido ser llevada a cabo. Al declarar la función se debe poner previamente la etiqueta **[ClientRpc]** en la línea anterior y empezar con **Rpc** en el nombre de la función. La Figura 67 muestra un ejemplo de este tipo de funciones:

```
[ClientRpc]
void RpcRespawn()
{
    if (!isLocalPlayer)
        return;

    Vector3 spawnPoint = Vector3.zero;

    if (spawnPoints != null && spawnPoints.Length > 0)
        spawnPoint = spawnPoints[Random.Range(0, spawnPoints.Length)].transform.position;

    transform.position = spawnPoint;
}
```

Figura 67. Función ClientRpc

Mediante la función mostrada en la Figura 67, el servidor puede hacer que cuando un jugador muera, este reaparezca en uno de los puntos señalados en el mapa.

- **TargetRPC:** Estas funciones realizan la misma tarea que las ClientRPC explicadas anteriormente, pero a diferencia de ellas, estas son ejecutadas desde el servidor a un único cliente de entre todos. Por lo tanto, mientras ClientRPC sirve para todos los clientes, mediante TargetRPC podemos especificar exactamente que cliente queremos que ejecute la función. Al declarar la función se debe poner previamente la etiqueta **[TargetRPC]** en la línea anterior y empezar con **TargetDo** en el nombre de la función. En la función declarada como TargetRPC se debe especificar en el primer parámetro de entrada el objeto *NetworkConnection* del jugador en concreto al cual afectará la función. Como ya se ha dicho anteriormente, esta herramienta no ha sido necesaria utilizarla en la realización de este proyecto.

5.1.4 Variables sincronizadas

Son aquellas que permiten ser sincronizadas por todos los jugadores de la partida. Para poder modificar variables utilizando las RPC mencionadas anteriormente, estas variables deben ser declaradas como variables sincronizadas.

Para declarar una variable que se sincronice en toda la red se debe colocar la etiqueta ***[SyncVar]*** antes de declarar la variable. La Figura 68 muestra dos ejemplos de variables sincronizadas:

```
[SyncVar(hook = "OnChangeHealth")]
public int currentHealth = maxHealth;

[SyncVar]
public int currentLive = maxLive;
```

Figura 68. Variables sincronizadas

La variable sincronizada *currentHealth* tiene un *hook* que representa una función. Cuando la variable *currentHealth* cambie a un nuevo valor, la función *OnChangeHealth* se ejecutará. La variable *currentLive* no tiene función asignada. El valor de esta variable está inicializado a *maxLive* para todos los jugadores, por lo tanto, todos los jugadores iniciarán con 5 vidas.

Las variables sincronizadas también pueden ser una lista de valores de un determinado tipo. En el proyecto se ha utilizado una lista sincronizada con los nombres de los jugadores que han creado una sala de streaming. Para ello a la hora de declarar la variable se debe poner que es del tipo ***SyncListString.*** La Figura 69 muestra cómo se declara esta lista sincronizada:

```
public SyncListString playerFree;
```

Figura 69. Lista sincronizada

Mediante una función RPC, estas listas pueden ser modificadas para todos los clientes. La función que modifica la lista definida en la Figura 69, se representa en la Figura 70:

```
[ClientRpc]
private void RpcPlayersFree()
{
    GameObject[] players = GameObject.FindGameObjectsWithTag("Player");

    foreach(GameObject player in players)
    {
        if (player.GetComponent<PlayerCode>().IsSending() && !player.GetComponent<PlayerCode>().IsBusy())
        {
            if (!playerFree.Contains(player.GetComponent<PlayerCode>().GetName()))
            {
                playerFree.Add(player.GetComponent<PlayerCode>().GetName());
            }
        }else
        {
            if (playerFree.Contains(player.GetComponent<PlayerCode>().GetName()))
                playerFree.Remove(player.GetComponent<PlayerCode>().GetName());
        }
    }
}
```

Figura 70. Uso de ClientRPC con la lista sincronizada

Como se ha dicho anteriormente, mediante este código se puede hacer un seguimiento de los jugadores que han creado una sala de streaming. Como se puede ver en el código, aquellos jugadores cuya variable ***isSending*** es *true* (han creado una sala) y cuya variable ***isBusy*** sea *false* (no se encuentran comunicándose con ningún jugador) serán añadidos a la lista ***playerFree***. Esta lista será utilizada para que los demás jugadores puedan consultar su lista de streaming activos y unirse a alguna sala.

5.2 Transmisión multimedia en tiempo real o Streaming en Unity

Después de incorporar la funcionalidad multijugador en el juego, se ha decidido añadir funciones de *streaming* para que los jugadores puedan comunicarse entre ellos por medio sus cámaras y micrófonos. Como ya se ha comentado, Unity carece de funcionalidades para esta tarea, por lo tanto, se ha tenido que hacer uso de un *asset* que ofrece la posibilidad de emplear **WebRTC** con Unity. El nombre de este *asset* es ***WebRTC Network***.

WebRTC Network es un *plugin* que permite conectar directamente dos equipos con Unity y que permite enviar mensajes “Reliables/Unreliables” usando los WebRTC Datachannels. El plugin solo permite enviar mensajes utilizando esos 2 canales, porque está diseñado para dar soporte a comunicación peer to peer, por lo tanto, se realizará el proyecto en virtud a usar únicamente los canales que proporciona el *asset*.

Este *asset* solo fue diseñado para el envío de mensajes de texto a través de sus clientes, por lo que se ha tenido que adaptar también para el envío de vídeo y audio a tiempo real. Al descargar el plugin, este venía con un servidor de señalización local que ha sido utilizado para depurar los errores durante el proceso de desarrollo.

Aunque se puede usar este servidor local gracias a **Node.js**, el *plugin* te permite conectar con un servidor público accesible a través de Internet por el cual se puede llevar a cabo la señalización. Además, ofrece dos servidores STUN para poder trabajar desde equipos que usan NAT o Firewalls. El servidor local se tuvo que modificar con *Eclipse* para permitir que se ejecutara la conexión.

Para realizar la conexión en Unity, se ha creado un script llamado ***CallApp***, el cual se adjunta al *GameObject* del jugador. En la Figura 44 del capítulo 4.5 se muestra este script añadido al *GameObject* del jugador.

5.2.1 *Script* CallApp

Este *script* es el encargado de hacer la conexión entre Unity3D y el servidor de señalización, y posteriormente enviar y recibir los datos pertinentes al *streaming*. Desde Unity3D se puede ver este *script* representado en la Figura 71.

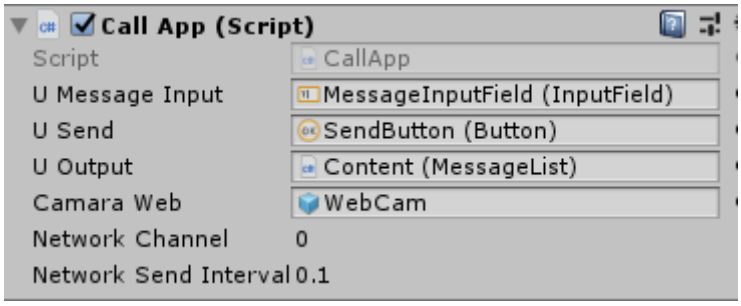

Figura 71. Componente CallApp

- ***Message Input:*** Este parámetro de entrada corresponde al campo de entrada del texto del chat. Mediante este parámetro, el *script* recoge el

mensaje que el jugador haya escrito en el campo y luego lo codifica a UTF8 para enviarlo a través de la red al jugador destino.

- ***Send:*** Es el botón de envío. Este *script* añade el comportamiento de envío al botón mediante la función ***SendButtonPressed***.
- ***Output:*** Un *script* que proporciona funciones propias del chat. Este *script* ya venía con el *asset* descargado.
- ***Camara Web:*** Es la textura 2D que visualizará los datos obtenidos de la red. Esta textura se encuentra en la interfaz de usuario de cada jugador.

El *script* internamente tiene parámetros privados que no pueden ser modificados exteriormente. Es importante señalar estos tres parámetros en la Figura 72:

```
#region CONECTION
//private string uSignalingUrl = "ws://localhost:12776/callapp";
private string uSignalingUrl = "wss://because-why-not.com:12777/callapp";
private string uIceServer = "stun:stun.l.google.com:19302";
private string uIceServer2 = "stun2.l.google.com:19302";
```

Figura 72. Direcciones usadas para WebRTC

La dirección de la variable ***uSignalingUrl*** que no se encuentra comentada, es la de la dirección del servidor de señalización público. La variable comentada es la del servidor local que se ha utilizado para depurar el funcionamiento de la conexión. La señalización se realiza mediante el protocolo ***WebSockets***.

5.2.3 Estados de comunicación de los jugadores

Antes de entrar en profundidad en el proceso de señalización que se estudiará en el apartado 5.2.4, se presentan las principales variables involucradas y cómo se gestiona el reconocimiento de jugadores activos que pueden iniciar una conversación. Se definen tres tipos de estados para los jugadores en función de su disponibilidad para iniciar una conversación con otro jugador:

- **Estado Activo:** El jugador ha iniciado una sala de *streaming*, esta sala tiene su nombre y él es el dueño de ella. Este estado se representa mediante un punto de **color verde**.

- **Estado Ocupado:** Simboliza que el jugador ya se encuentra dentro de una sala de *streaming*, pero a diferencia del estado Activo, en el Ocupado el jugador ya está comunicándose con otro. Este estado se representa mediante el **color rojo**.
- **Estado Desocupado:** El jugador ni ha iniciado una sala de *streaming*, ni está dentro de una. Este estado se representa mediante el **color amarillo**.

Cuando un jugador esté en el estado Desocupado y presione la tecla F1 (imaginemos que lo hace un jugador llamado el *Player1*), hará una conexión con el servidor de señalización y creará una sala de *streaming*, la cual tendrá el nombre del jugador que la ha creado. La función que se encarga de realizar ese proceso se llama ***OnClickJoinStreaming,*** y se encuentra dentro del *script CallApp*.

Una vez que el jugador ha creado una sala *streaming*, cambiará una variable sincronizada *booleana* contenida en *PlayerCode* llamada ***isSending.*** Esta variable tendrá el valor *true*, lo que significa que el jugador está emitiendo una sala de *streaming*, y, por lo tanto, el jugador estará en el **estado Activo**. En la Figura 73 se aprecia como en el momento de que el jugador crea esa sala, modifica el valor de *IsSending*.

```
case NetEventType.ServerInitialized:
    {
        mIsServer = true;
        address = evt.Info;
        Debug.Log("Server started. Address: " + address);
        Debug.Log("ConnectionId: " + evt.ConnectionId);
        GetComponent<PlayerCode>().CmdIsSending();
    }
    break;
```

Figura 73. Modificación del valor isSending

Como se puede apreciar en la Figura 73, esta modificación del valor se realiza mediante una función *Command.* De esta manera el cambio de la variable será perceptible por todos los jugadores de la partida, y verán a este jugador con su punto de color verde indicando que está a la espera de que alguien entre a su sala. En la Figura 74 aparece el jugador con su punto de color verde.

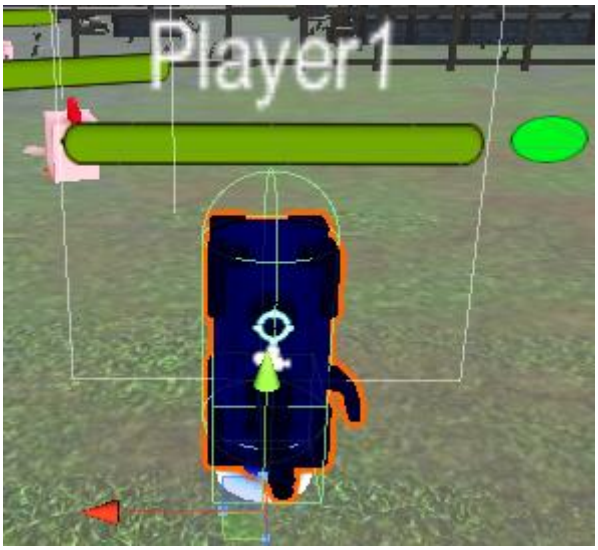

Figura 74. Jugador en estado "Activo"

Al mismo tiempo, un jugador cualquiera de la partida podrá ver si hay salas de *streaming* abiertas por parte de otro jugador. Para hacer esto, el jugador tendrá que presionar sobre la tecla ***Ctrl* izquierda** y aparecerá un panel llamado ***Streaming List*** donde aparecerá quien está con su sala creada y en espera en ese momento.

Para que el panel muestre esa información, en el momento de presionar *Ctrl,* se lee el valor *isSending* de todos los jugadores de la partida. Aquellos que tengan el valor de esta variable a *true*, simbolizará que han iniciado una sala, y están esperando a que alguien se una a ellos. En la Figura 75 se ve el fragmento de código que se encarga de realizar esta comprobación, el cual se encuentra añadido al *GameObject Call Manager* del que ya se habló en el capítulo 3.

```
[ClientRpc]
private void RpcPlayersFree()
{
    GameObject[] players = GameObject.FindGameObjectsWithTag("Player");

    foreach(GameObject player in players)
    {
        if (player.GetComponent<PlayerCode>().IsSending() && !player.GetComponent<PlayerCode>().IsBusy())
        {
            if (!playerFree.Contains(player.GetComponent<PlayerCode>().GetName()))
            {
                playerFree.Add(player.GetComponent<PlayerCode>().GetName());
            }
        }else
        {
            if (playerFree.Contains(player.GetComponent<PlayerCode>().GetName()))
                playerFree.Remove(player.GetComponent<PlayerCode>().GetName());
        }
    }
```

Figura 75. Código para comprobar el estado de los jugadores

Cuando el *Player2* presiona la tecla *Ctrl*, aparecerá lo siguiente que se muestra en la Figura 76:

Figura 76. Streaming List

Como se ve en la Figura 76, aparece en la lista que el *Player1* ha iniciado una sala de *streaming*. Por lo tanto, el *Player2* puede unirse a ella haciendo *click* en el nombre del jugador (el *Player1* no será capaz de ver su sala abierta ya que no tiene sentido que él pueda unirse a su propia sala).

Como es lógico, puede haber más de un jugador con salas de streaming abiertas. Se permite hasta un máximo de 6 salas, ya que el valor coincide con el máximo número de jugadores permitidos en una misma partida.

Figura 77. Varias salas de streaming abiertas

En el momento que un jugador accede a una sala de *streaming* para estar con otro jugador, ambos estarán comunicados. Además, ambos estarán en el modo **Ocupado**, lo que se traduce en que les aparecerá un indicador de **color rojo**. Esto sucede porque cuando dos usuarios se encuentran conectados a una misma sala, se establece a ***true*** una variable llamada ***IsBusy*** para cada jugador.

De la misma forma que en la Figura 73 cuando se pone a true ***isSending***, en la siguiente Figura 78 se muestra que fragmento del código de *CallApp* modifica el valor de ***IsBusy*** a true.

```
case NetEventType.NewConnection:
    {
        mConnections.Add(evt.ConnectionId);
        GetComponent<PlayerCode>().CmdIsBusy();

        if (!mIsServer)
        {
            Append(DateTime.Now.ToString()+ ": Te has unido a una sala.");
            isConnectedAsClient = true;
        }
        else
        {
            Append(DateTime.Now.ToString()+ ": Un jugador se ha unido a tu sala.");
        }
    }
    break;
```

Figura 78. Modificación del valor isBusy

Cuando esto sucede, aparecerá un punto de color rojo en el jugador que tenga esta variable a *true*. En la siguiente Figura 79 se muestra este suceso:

Figura 79. Jugador en modo "Ocupado"

En el caso de que un jugador no esté dentro de una sala de *streaming* ni la haya creado, aparecerá con un punto de color amarillo. Esto se traduce a que

las variables *isSending* y *IsBusy* son *false*. En la siguiente Figura 80 se muestra este suceso:

Figura 80. Jugador en modo "Desocupado"

5.2.4 Señalización del plugin WebRTC Network

Se ha visto en los apartados anteriores qué variables se modifican dentro del juego para que los jugadores sean conscientes de a qué usuarios pueden o no conectarse. Todo este proceso de comprobar en qué estado se encuentra el jugador ha sido desarrollado enteramente en Unity3D usando las herramientas que proporciona la API de *Networking* para actualizar modificar valores booleanos (las RPC concretamente). Pero hasta el momento no se ha visto cómo funciona internamente el *asset* de *WebRTC Network*.

Cuando un jugador crea una sala de *streaming*, crea una instancia del tipo ***WerbRTCNetworkFactory***. Mediante la función ***StartServer***, crea un canal de comunicación cuyo nombre será el nombre del jugador que crea la sala. La Figura 81 muestra esta función:

```
mNetwork.StartServer(GetComponent<PlayerCode>().GetName().ToString());
```

Figura 81. Función de creación de una sala de streaming.

En ese momento se hace una conexión con el servidor de señalización para que cree una sala con el nombre del jugador que la ha creado. En ese momento, si el servidor de señalización se encuentra localmente, se puede apreciar en la Figura 82 qué proceso se realiza.

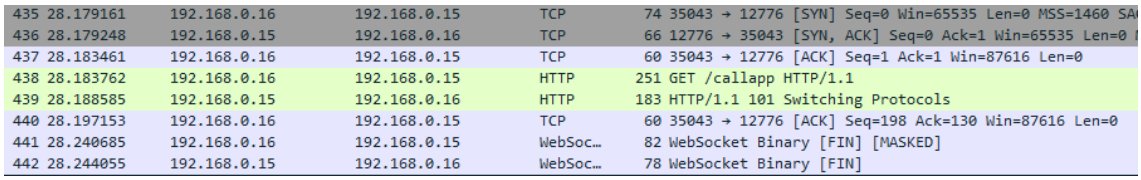

Figura 82. Proceso de señalización.

En la Figura 82 aparece una captura de Wireshark de la red durante la ejecución del juego. Aparecen dos direcciones IP que corresponde a un escenario de un PC y un dispositivo Android conectados a la misma red y jugando una partida:

- **192.168.0.15:** Corresponde a un jugador de la partida. Es el que hace de servidor de la partida y es además la dirección IP donde está el servidor de señalización.
- **192.168.0.16:** Es el dispositivo Android de la red que se conecta como cliente al juego.

Paralelamente a la señalización de la Figura 82, ya que en la misma máquina donde funciona el servidor de señalización hay un jugador también que está jugando, se puede ver como Unity intercambia paquetes UDP para hacer que se sincronice el juego a lo largo de la red. En el caso de la información referente a la transmisión en tiempo real, esta viaja sobre UDP utilizando SCTP/DTLS. En la Figura 83 se muestra los paquetes referentes tanto a los UDP intercambiados por Unity y los datos cuando se encuentran dos jugadores comunicándose y jugando. Además, podemos ver cómo hacer uso de un servidor STUN para identificar las direcciones IP públicas de los usuarios que participan en la conversación al inicio de la misma.

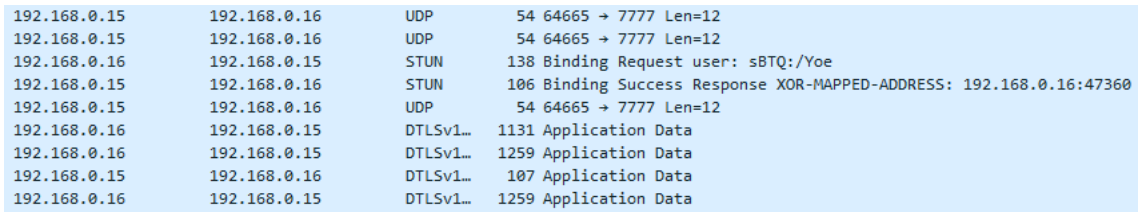

Figura 83. Intercambio de datos durante la partida y la videoconferencia.

En primer lugar, se explicarán líneas correspondientes al protocolo **HTTP** y al protocolo **WebSocket** de la Figura 82, los cuales son utilizados para realizar la señalización.

Si abrimos la línea del protocolo HTTP en donde hace el GET podemos ver la siguiente información.

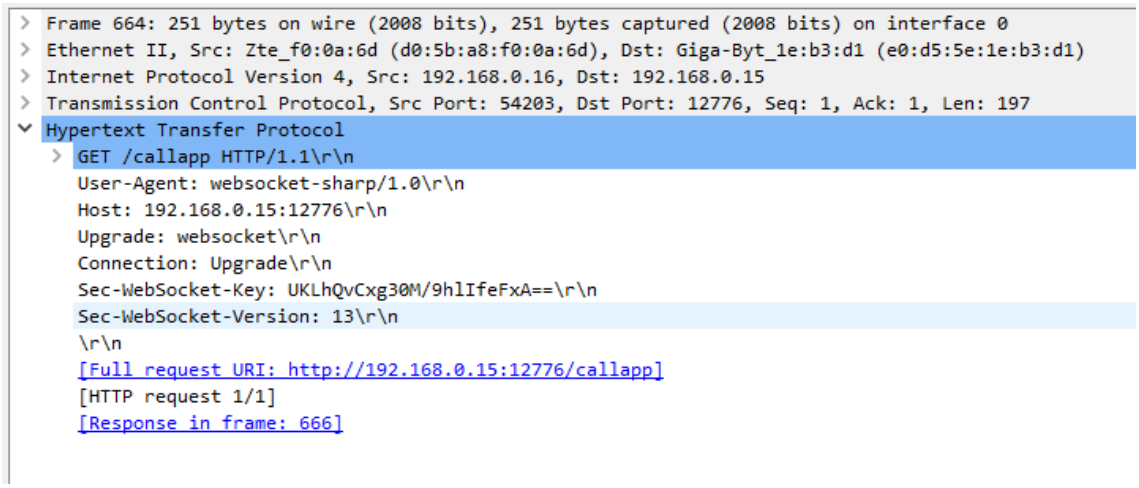

Figura 84. GET de HTTP

En la Figura 84 se puede ver como se hace una petición GET a la dirección IP donde alberga el servidor de señalización. En ella, se le solicita al servidor de señalización el deseo de abrir una conexión WebSocket. A continuación, el servidor de señalización responderá a esta petición en la Figura 85.

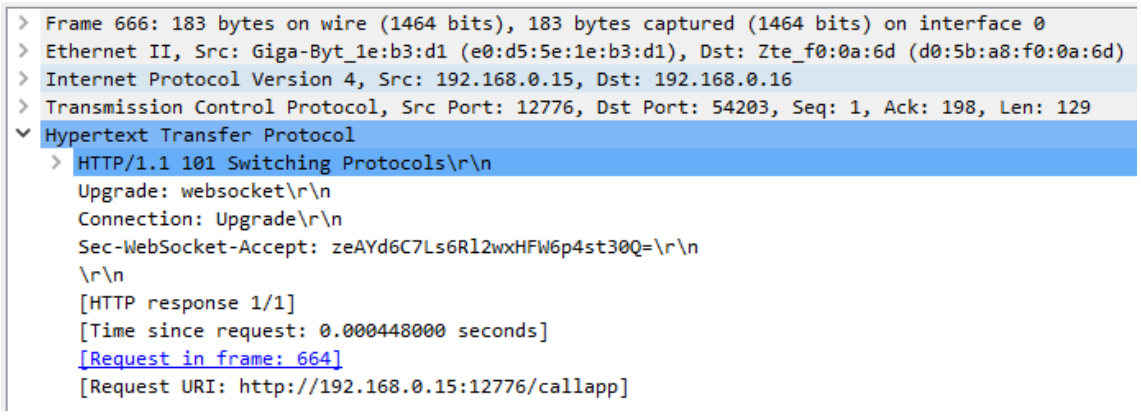

Figura 85. Respuesta del servidor de señalización

En la Figura 85, el servidor de señalización acepta la petición que le ha llegado y envía su respuesta de aceptación.

Posteriormente en la Figura 86 se muestra como el servidor de señalización crea la conexión WebSocket correctamente. A continuación en la Figura 86 se muestra esa información deglosada:

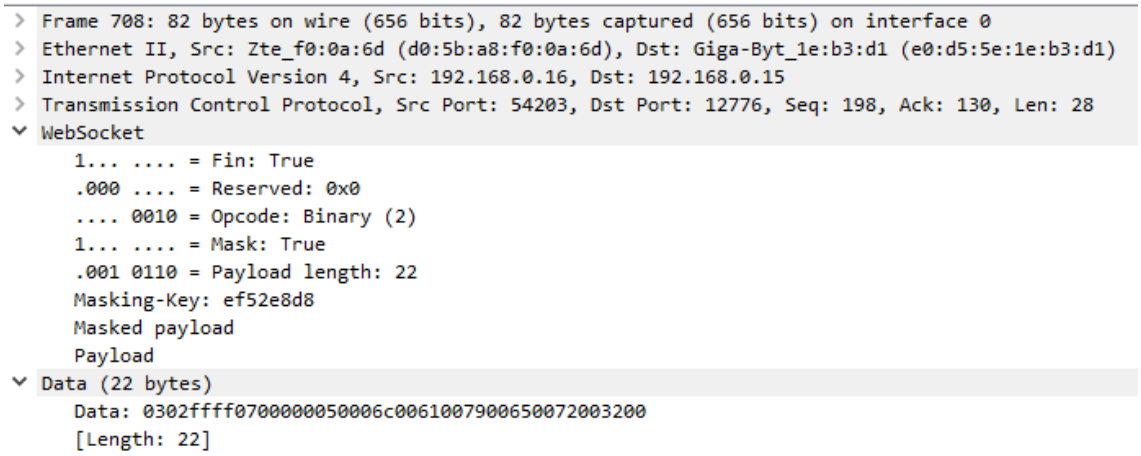

Figura 86. Creación de una conexión WebSocket

Si depuramos el proceso de creación de una sala de streaming en el servidor de señalización local, aparecerá el siguiente mensaje:

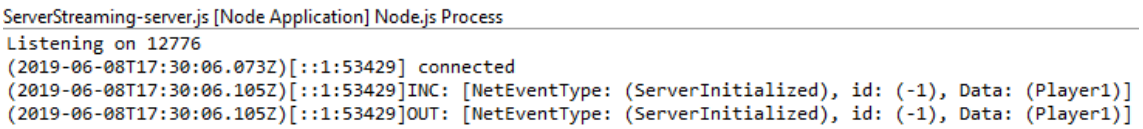

Figura 87. Creación de la sala en el servidor de señalización.

Una vez creada la conexión, Unity3D recibirá un mensaje el tipo ***ServerInitialized*** informando de que la sala ha sido creada correctamente. En ese momento, tal y como se vio en la Figura 73, el valor de *IsSending* se establece a true.

Posteriormente, un jugador en **modo Desocupado** se une a la sala mediante el panel *Streaming List* visto anteriormente. En ese momento, el servidor de señalización hace una conexión entre los dos usuarios haciendo la negociación correspondiente. En la Figura 88 se muestra dicha negociación.

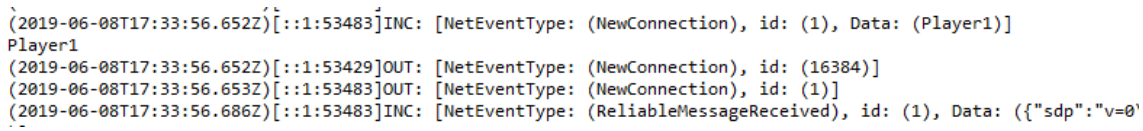

Figura 88. Unión de un jugador a la sala

Se puede apreciar como hay un intercambio de parámetros mediante el uso de SDP. A continuación, se verá los parámetros más relevantes que tiene los campos SDP intercambiados y cuál es su sentido.

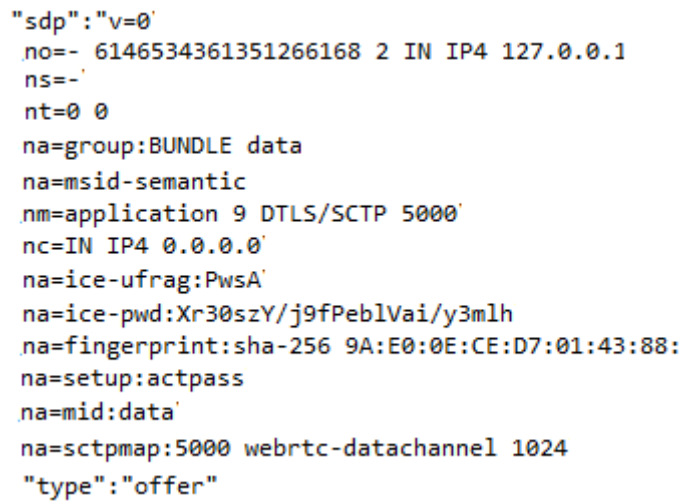

Figura 89. SDP intercambiado

Podemos comprobar en la Figura 89 que el campo ***nm*** establece una comunicación **DTLS/SCTP 5000.** Este parámetro indica que la para la comunicación se utiliza el protocolo *Stream Control Transmission Protocol (*SCTP) encima de *Datagram Transport Layer Security Protocol*[19] (DTLS). [19]

SCTP sobre DTLS se utiliza para intercambiar datos **no multimedia** en el protocolo WebRTC. Aquí se puede apreciar como el *asset* solo viene preparado para el envío de mensajes de texto entre los usuarios (no para datos multimedia), pero gracias a las funciones que trae Unity3D para codificar y descodificar el vídeo y el audio en bytes, se puede a nivel de aplicación recibir los datos intercambiados en bytes para posteriormente transformarlos en audio o vídeo. De esta manera, se evita modificar el asset gratuito y aprovechar el tipo de comunicación. Además, si se modificase el asset luego no podríamos utilizar el servidor de señalización público ya que no estaría preparado para los cambios. El campo `na=sctpmap:5000 webrtc-datachannel 1024` nos da a entender que se hace uso de *Datachannels* en la conexión entre pares *PeerConnection*.

[19] https://tools.ietf.org/html/rfc6347

Una vez que este proceso se ha completado, ambos jugadores (*Player1* y *Player2*) estarán unidos a una misma sala de *streaming*, y aparecerán en el modo Ocupado.

5.2.5 Problemática para una conexión multipunto de streaming o flujo continuo de datos multimedia.

Cuando un jugador crea o se conecta a una sala de *streaming*, internamente se llena una lista del tipo ***ConnectionId*** llamada ***mConnections***, la cual por cada nuevo cliente que se une a la sala aumenta en uno el tamaño de la lista cuyo parámetro llamado ***id*** incrementa de manera secuencial desde 1 hasta el número de clientes que se encuentren en la sala. De modo, que si al creador de una sala (llamémoslo *Player1*) se le unieran dos jugadores (*Player2* y *Player3*, por ese orden), el *Player2* sería identificado por el *Player1* con la **id = 1**, y el *Player3* sería identificado con la **id = 2**, por lo tanto, la lista ***mConnections*** tendría un tamaño de dos.

Imaginemos un escenario, en donde hay dos jugadores jugando una partida; estos son *Player1* y *Player2*. El jugador *Player1* desea abrir una sala de *streaming* presionando la tecla F1, y establece la comunicación que se ha mostrado en el capítulo anterior. Cuando la sala se crea, el jugador *Player2* puede unirse a ella consultando su lista de *streaming* activos. Tan solo debe hacer *click* en el nombre de la sala del *Player1* y ya se habría unido a dicha sala. Cuando la conexión se ha hecho, internamente cada jugador añade un espacio a su lista mConnections y cada uno guarda **localmente** el valor de ***id* = 1** para identificar la conexión del otro. En la Figura 90 aparece este escenario:

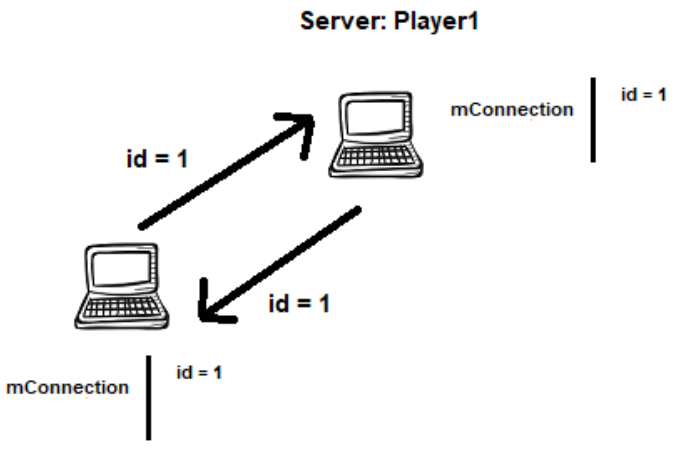

Figura 90. Sala de streaming entre dos usuarios

Si un jugador quiere enviar datos al otro, simplemente deberá hacer uso de la función ***SendData*** que el *plugin* de WebRTC Network incorpora. Por ejemplo, la Figura 91 muestra cómo se envía la imagen de la cámara al usuario destino.

```
foreach (ConnectionId id in mConnections)
{
    mNetwork.SendData(id, msgData, 0, msgData.Length, false);
}
```

Figura 91. Función para enviar datos por la red

Como se puede ver, se recorre el vector ***mConnections*** el cual guarda las conexiones activas localmente (se debe recordar que las id se generan los localmente) para la sala. Ya que, en este ejemplo, cada uno reconoce al otro como id = 1 (ya que es una variable local que el propio plugin asigna localmente cuando se crea una conexión) se enviará los datos en *bytes* de la imagen al usuario con id = 1 (en este caso al *Player1* o al *Player2,* dependiendo de a quién pertenezca ese código).

¿Qué pasaría si hubiera más de un usuario? La Figura 92 muestra el mismo escenario, pero esta vez con un usuario más añadido a la sala.

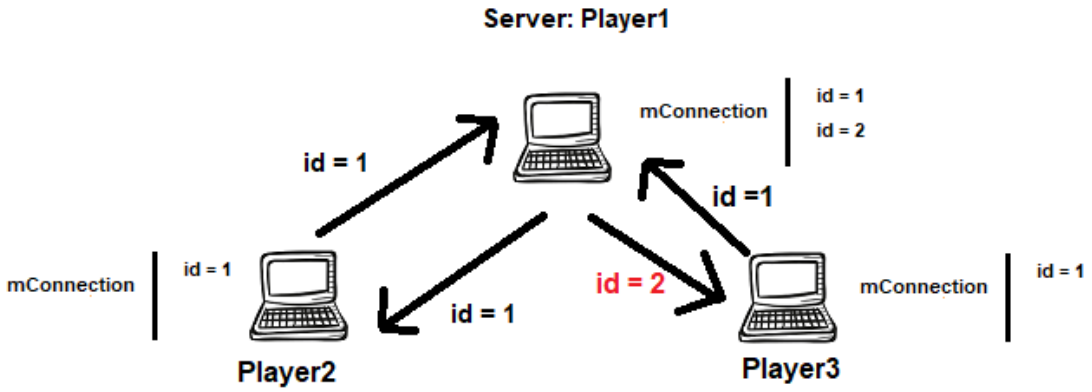

Figura 92. Problemática con unas salas de streaming con 3 usuarios

Como se puede comprobar el tercer usuario (el *Player3*), se conecta a la sala creada por el *Player1*. El *Player1* guardará la id del nuevo usuario (que tendrá el valor de 2) y el *Player3* que se acaba de unir guardará la *id* del *Player1* como con el valor 1. El *Player2 no* es capaz de reconocer la existencia del *Player3,* esto se debe a que como es una conexión P2P, no está preparada para albergar más usuarios y requeriría cambiar su distribución.

5.2.6 Envío y recepción de imágenes en WebRTC Network

Como ya se ha mencionado anteriormente, los datos viajan a través de la red en forma de *bytes* mediante DTLS/SCTP el cual se utiliza para enviar datos no multimedia. La compresión de los datos se hace a **nivel de aplicación**, por medio de las herramientas que proporciona Unity3D. Es importante saber que lo normal a la hora de hacer una transmisión de vídeo es utilizar funciones de codificación sobre RTP (H.264 en el caso del vídeo). Sin embargo, aquí no se utiliza.

El funcionamiento consiste en coger los fotogramas que se obtienen de la cámara y enviarlos a través de la red como imágenes independientes. Ya que se cogen aproximadamente 25 fotogramas por segundo, cuando estas imágenes lleguen al receptor se irán mostrando en su interfaz de usuario de tal forma que se verá con fluidez como si fuera un vídeo. En la Figura 93 se muestra el fragmento de código donde se realiza la extracción de los pixeles de cada imagen y su correspondiente conversión a *bytes*.

```
private void SendVideoInBytes()
{
    if (mNetwork != null && GetComponent<PlayerCode>().IsBusy())
    {
        currentTexture.SetPixels(webCam.GetPixels());
        currentTexture.Apply();
        byte[] pngBytes = currentTexture.EncodeToJPG(65);

        foreach (ConnectionId id in mConnections)
        {
            mNetwork.SendData(id, pngBytes, 0, pngBytes.Length, false);
        }
    }
}
```

Figura 93. Compresión a JPG

Mediante la función ***GetPixels*** se extraen los pixeles de la cámara que se obtienen en un momento concreto cuando esta se encuentra funcionando, y luego se insertan a una textura del tipo *Texture2D* (una imagen). Después la función ***EncodeToJPG*** transforma la imagen 2D a una secuencia de *bytes*. El valor entre paréntesis (aparece como 65 en la Figura 93) es el valor de la calidad con la que se comprimirá la imagen convertida en *bytes*. Se ha escogido un valor no demasiado elevado para que no repercuta negativamente en la calidad de la imagen, pero que además tampoco cargue

en exceso el tráfico de red. Finalmente, mediante la función ***SendData*** se envía los *bytes* de la imagen a los usuarios correspondientes recorriendo el vector *mConnections* como ya se explicó anteriormente.

Para la recepción, el usuario recibe el flujo de *bytes* pertenecientes a las imágenes por el canal ***"UnReliable"*** (hay dos canales disponibles en el *asset* de *WebRTC Network*, y sólo han sido necesarios utilizar esos dos para el proyecto).
Simplemente se debe extraer los *bytes* del mensaje que le llega al receptor y pintarlos mediante la función ***LoadImage*** en una textura que se encuentra en la interfaz del usuario del jugador que recibe la imagen. El proceso de pasar los *bytes* a una imagen lo gestiona esa función. En la Figura 94 se muestra este proceso en el código:

```
MessageDataBuffer buffer = (MessageDataBuffer)evt.MessageData;
byte[] data = evt.GetDataAsByteArray();

if (data != null && data.Length > 250)
{
    timerToggleTexture = 3.0f;
    getTexture.LoadImage(data);
    getTexture.Apply();
    CamaraWeb.GetComponent<Image>().enabled = true;
    CamaraWeb.GetComponent<Image>().material.mainTexture = getTexture;
}
```

Figura 94. Recepción de los datos de vídeo

5.2.7 Envío y recepción de los datos de audio en WebRTC Network

En el caso del audio el proceso es algo más complejo. En el momento en el que se inicia la comunicación, Unity va recogiendo internamente las muestras de audio obtenidas del micrófono utilizando la clase **Microphone**. Como se puede ver en la Figura 95, se hará uso de la función Start que provee la clase Microphone para iniciar esta recogida de datos, además, también se ha especificado la frecuencia de muestreo la cual será de 16kHz (no se ha elegido una frecuencia más alta para no comprometer a la red). La variable **microphone** hará referencia por tanto al micrófono por defecto que Unity detecte que está conectado al PC.

```
if (Microphone.devices.Length != 0)
{
    microphone = Microphone.Start(nameDevice, true, 10, FREQUENCY_RATE);
    isSpeaking = true;
}
```

Figura 95. Activación de un micrófono

Posteriormente en la función **SendAudioBytes** se procederá a obtener las muestras de la variable *microphone* y enviarlas a través de la red. En la Figura 96 se muestra el fragmento de código de dicha función que hará esta tarea.

Las muestras que se van almacenando internamente se guardan en un vector interno con un tamaño de 160.000 muestras, que cuando lleguen a este valor se reinicializa de nuevo a 0. Mediante la función ***GetPosition***[20] se obtiene la posición actual del número de muestras recogidas por el micrófono con un retardo de 30ms.

Una vez que se obtiene el número de muestras almacenadas en ese vector interno, se procederá a obtenerlas mediante la función ***GetData*** cuyo tamaño de muestras a recoger será la diferencia entre la posición actual del vector interno (variable *pos*) y la posición guardada en la última iteración (variable *lastPos*). Finalmente, el vector obtenido será del tipo *float* y se deberá convertir a otro vector de tipo *bytes* mediante la función ***ToByteArray***. De esta manera, podremos enviar los datos a través de la red.

[20] GetPosition: Obtiene la posición en muestras de la grabación. Por defecto se recoge muestras cada 30ms.

```
private void SendAudioBytes()
{
    if (mNetwork != null && GetComponent<PlayerCode>().IsBusy())
    {
        if ((pos = Microphone.GetPosition(null)) > 30)
        {
            if (lastPos > pos)
                lastPos = 0;

            if ((pos - lastPos) > 0)
            {
                float[] sample = new float[((pos - lastPos) * microphone.channels)];
                microphone.GetData(sample, lastPos);
                byte[] bytesFromAudio = ToByteArray(sample);

                lastPos = pos;

                foreach (ConnectionId id in mConnections)
                {
                    mNetwork.SendData(id, bytesFromAudio, 0, bytesFromAudio.Length, true);
                }
            }
        }
    }
}
```

Figura 96. Recogida y envío del audio

Para la recepción, al igual que con las imágenes, se extraen los *bytes* del mensaje que llega desde la red. El siguiente paso será volver a transformar el vector de *bytes* a otro de tipo *float* para que pueda ser reconocible por Unity3D como muestras de audio. En la Figura 97 aparece este proceso:

```
MessageDataBuffer buffer = (MessageDataBuffer)evt.MessageData;
byte[] data = evt.GetDataAsByteArray();
float[] audioData = ToFloatArray(data);

for (int i = 0; i < audioData.Length; i++)
{
    read.Add(audioData[i]);
}

if (data != null && read.Count >= FREQUENCY_RATE )
{
    AudioSource audioSource = GetComponent<AudioSource>();
    speakers.SetData(read.ToArray(), 0);
    audioSource.clip = speakers;
    audioSource.Play();
    read.Clear();
}
```

Figura 97. Reproducción del audio recibido

Mediante la función ***GetDataAsByteArray*** se obtiene los datos que llegan desde la red en forma de bytes. Luego se convertir a un vector del tipo float mediante la función ***ToFloatArray.***

Para reproducir las muestras de audio en el receptor se deberá hacer uso del componente **AudioSource** el cual se encuentra adherido al jugador. En primer lugar, se copiarán los datos obtenidos en otro **AudioClip** llamado ***speakers***. Posteriormente, el componente AudioSource copiará los datos de *speakers* (es la única forma de que el componente pueda leer clips de audio) y posteriormente los reproducirá mediante la función Play.

Es importante saber que cuando se extraen los datos de las muestras del micrófono mediante la función *GetData* en la Figura 96, el número de muestras obtenidas depende del bucle de actualización de Unity, por lo tanto, se consigue un número de muestras muy pequeño y se hace necesario que estas sean acumuladas antes de ser reproducidas en el receptor. Por lo tanto, para solucionar este problema se ha decido crear un pequeño buffer que almacene las muestras en el receptor y las reproduzca cuando haya un número de muestras considerable, de esta manera no se escuchará el sonido entrecortado.

5.2.7 Envío y recepción de los mensajes en WebRTC Network

En el caso del envío de los mensajes de texto la solución ha sido más sencilla, ya que *WebRTC Network* sí está preparado para enviar mensajes de este tipo. El *plugin* ya implementa una ventana de interfaz que servirá como chat, a la cual se le ha tenido que retocar el color y la funcionalidad del botón enviar. En la siguiente Figura 98 se muestra dicha ventana:

Figura 98. Ventana del chat

Para enviar el mensaje, el emisor escribe el contenido dentro del campo de escritura y presiona la tecla *Send*. La tecla *Send* recogerá la cadena de texto escrita y la guardará en una variable para posteriormente codificarla a ***UTF8*** y poder enviarla a través de la red. En la Figura 99 se muestra este proceso:

```
byte[] msgData = Encoding.UTF8.GetBytes(msg);
foreach (ConnectionId id in mConnections)
{
    mNetwork.SendData(id, msgData, 0, msgData.Length, false);
}
```

Figura 99. Codificación de la cadena de texto a UTF8

Posteriormente se envía el mensaje al usuario conectado al otro extremo de la misma que con el caso del vídeo y el audio. El texto se envía por el canal ***"Unreliable"*** que ofrece el *plugin* de Unity3D. Se debe recordar que por este canal también iba la información del vídeo, por lo tanto, para que el receptor sepa diferenciar entre los mensajes de vídeo y texto se mira la longitud del mensaje. En la Figura 100 se hace esta comprobación:

```
if (data != null && data.Length > 250)
{
    timerToggleTexture = 3.0f;
    getTexture.LoadImage(data);
    getTexture.Apply();
    CamaraWeb.GetComponent<Image>().enabled = true;
    CamaraWeb.GetComponent<Image>().material.mainTexture = getTexture;
}
else if(data != null && data.Length <= 250 )
{
    string msg = Encoding.UTF8.GetString(buffer.Buffer, 0, buffer.ContentLength);

    if(mIsServer)
    {
        SendString(msg);
        Append(msg);
    }
    else
    {
        Append(msg);
    }
}
```

Figura 100. Recepción del mensaje y el vídeo por el canal "Unrealiable"

Al comprobar la longitud, si esta es inferior de 250, el mensaje enviado es de tipo texto, si es superior es un mensaje de vídeo. Los bytes pertenecientes al envío del vídeo siempre van a ser superiores a 250 por muy mala que sea la

calidad de la cámara, pero por si acaso, los usuarios tienen limitados el número de caracteres que pueden enviar a 120 caracteres.

Si el receptor recibe el mensaje y detecta que es del tipo texto, decodificará dicho mensaje para convertirlo en una cadena de texto y pintarla en su correspondiente ventana de *chat.*

6. Interfaces desarrolladas

Unity3D es multiplataforma, es decir, aporta facilidades para hacer que sus juegos sean desarrollados para varias plataformas. En este proyecto se ha optado por crear una versión para Android y otra versión que pudiese funcionar en las *Oculus Go*, además de la de PC descrita en capítulos anteriores.

Cada plataforma tiene su manera distinta de gestionar el movimiento del avatar, y la forma de interactuar con el juego, pero todos los demás componentes de Unity mostrados a lo largo de este documento se mantienen invariables.

La versión de Android, al igual que en la de PC, está preparada para permitir que el usuario pueda crear o entrar a una sala de *streaming*. Para la versión de las *Oculus Go* esta funcionalidad no está implementada, ya que supondría una carga innecesaria en la pantalla del jugador.

En este capítulo se explicará cómo se ha desarrollado una versión para Android y otra versión para las *Oculus Go*. Para ambas, es necesario que el proceso de compilación que Unity hace del juego se cambie al de la plataforma Android. Para ello, se debe ir a la opción ***File*** que se encuentra en la pestaña superior izquierda de la pantalla de trabajo, buscar a continuación la opción ***Player Setting*** y seleccionar la plataforma Android. Finalmente, se le daba dar a la opción ***Switch Plaform***. Cuando se haya esto, si se le da a opción ***Build***, Unity creará un ***.apk*** que será ejecutable tanto para un dispositivo móvil con Android como para las *Oculus Go*.

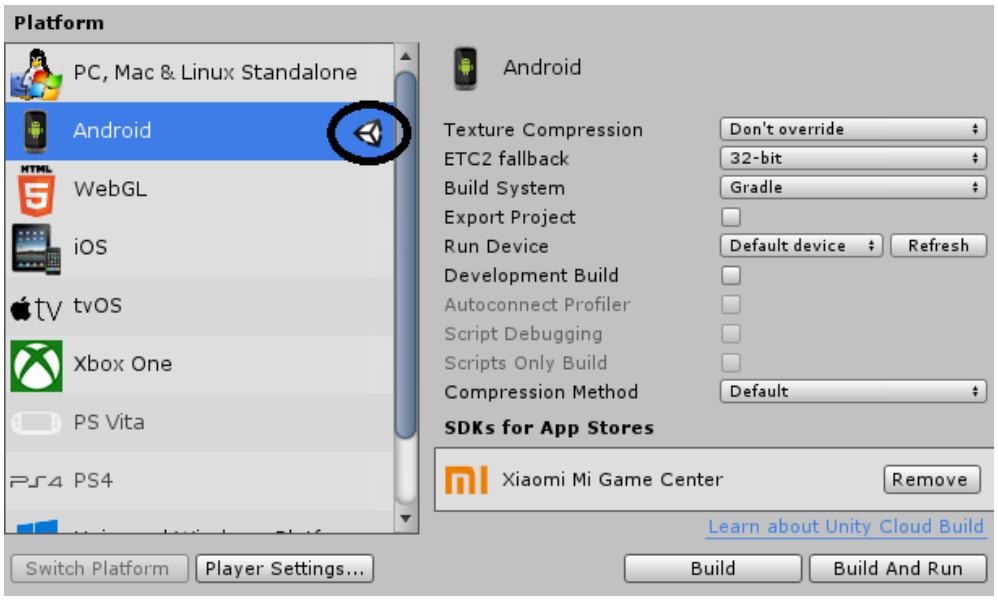

Figura 101. Lista de selección de plataformas

Otro paso importante que se debe realizar es colocar un nombre válido en el parámetro de entrada ***Package Name***, si no se hace esto Unity3D reportará un error cuando se intente compilar el juego. Para ir a la sección donde se encuentra este parámetro se debe acceder a ***Player Settings*** dentro de la pantalla que aparece en la Figura 101 anterior. Después se abrirá una ventana en la parte derecha de la pantalla.

Figura 102. Player Settings

A continuación, se deberá abrir el desplegable ***Other Setting*** y buscar el parámetro ***Package Name***. Finalmente, se debe colocar un nombre con el mismo formato que el de la Figura 103.

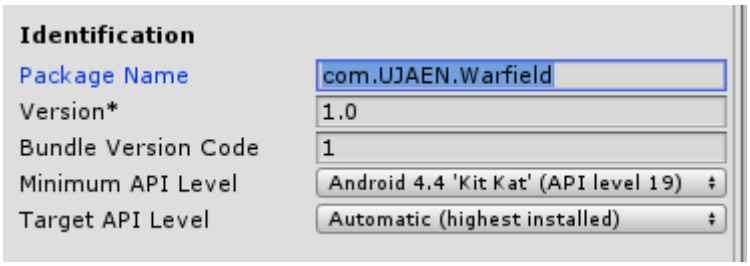

Figura 103. Valor Package Name

En la Figura 103, se ha seleccionado como ***Minimum API Level*** la 19, se tendrá que poner esta como mínimo, de lo contrario cuando se quiera compilar el proyecto con las *Oculus Go* reportará un error.

Previamente al cambio de plataforma, se debe especificar la ruta de los *SDK* que necesitará Unity3D para ejecutar la compilación Android. Los pasos para hacer esto aparecerán detallados en el apartado Anexos de este documento.

6.1 Compatibilidad con Android

La versión de Android se encuentra integrada en el mismo proyecto que la versión de PC. Para diferenciar cuando el juego se ejecuta en una plataforma u en otra, Unity tiene una función llamada ***Application.platform*.** Mediante una sentencia *if* se puede hacer una distinción entre los elementos que aparecerán en la pantalla entre una versión y otra, y, además, se podrá diferenciar la manera en la que el jugador podrá mover su avatar.

En el caso de que juego se ejecute en un ordenador, el control del avatar será a través de las teclas y el ratón. En el caso de Android, la interfaz de usuario cuenta con dos controles tipo *joystick* (uno a cada lado de la pantalla), un botón negro en la parte derecha inferior de la pantalla y unos botones de pulsación mostrados en la parte superior de la pantalla.

Con el *joystick* izquierdo el jugador podrá mover la cámara del avatar a través de su pantalla táctil, y con el *joystick* derecho podrá controlar el movimiento. El punto negro simboliza el botón para disparar el arma. Además, en la parte superior de la pantalla (debajo de la barra del control de salud) las tres teclas permitirán al jugador interactuar con las funciones del *streaming*. En la Figura

104 se muestra un ejemplo de cómo se vería el juego mientras esté funcionando en un dispositivo Android.

Figura 104. Interfaz de usuario en un dispositivo Android

En la Figura 104, las teclas que aparecen en la parte superior de la pantalla permiten al usuario poder unirse, crear y participar en una sala de *streaming* al igual que con un PC. Las teclas representadas en la pantalla son análogas que las que usan en el teclado en la versión de PC.

- **F1:** Crear una sala de streaming nueva.
- **F10:** Abandonar una sala de streaming.
- **F9:** Encender/Apagar el micrófono y la cámara.

El acceso a la funcionalidad de streaming se realiza de la misma forma que en el caso de la versión de PC. Es decir, se debe abrir la lista de *streaming* activos, funcionalidad que en este caso se ha implementado en la interfaz mediante una acción de "doble *tap" o doble pulsación* sobre esa porción de la pantalla. En la Figura 105 se muestra dónde se abre la ventana de streaming, y se representa en un cuadrado de color naranja la porción aproximada donde el jugador debe hacer **doble *tap*** para abrir dicha lista.

Figura 105. Lista de salas abiertas en Android

En la Figura 106 la porción de código que se encuentra en el *script Player Code* que realiza la función de adaptar la interfaz gráfica a esta última descripción (ver Figura 105) en el caso de que la ejecución se realice en una plataforma Android.

```
if (Application.platform == RuntimePlatform.Android)
{
    JoystickAndroid.SetActive(true);
    ClickShoot.SetActive(true);
    JoystickCamera.SetActive(true);
    ButtonForInitAndroid.SetActive(true);
    ButtonForExitAndroid.SetActive(true);
    ButtonForDevices.SetActive(true);
    isAndroid = true;
}
```

Figura 106. Código para adaptar la UI al teléfono

Este código se encuentra dentro de la función *Start* de *PlayerCode*, por lo tanto, solo se ejecutará una vez y será justo en el momento en el que jugador aparezca en el mapa de juego por primera vez.

6.1.1 Joysticks de movimiento de la cámara y el avatar

El movimiento de la cámara y el jugador se lleva a cabo mediante los *joysticks* que se implementan en la versión de Android. El modelado de los *joysticks* y el funcionamiento básico ha sido proporcionado por un *asset* de la *Asset Store* llamado ***UJoystick***. Este *asset* proporciona un *GameObject* llamado *JoystickMovement* con la funcionalidad de moverse al contacto con el dedo en una pantalla táctil. Mediante esa base, se ha podido añadir la funcionalidad de incluir movimiento al avatar cuando estos *joysticks* detecten movimiento en la pantalla.

Cada joystick se representa en Unity3D como un *GameObject*, el cual aparece en la siguiente Figura 107.

Figura 107. GameObject asociado al Joystick

El elemento *Center* representa el punto central del *joystick*, que servirá como referencia cuando se mueva el *Stick*. El Stick, es por lo tanto el círculo de color negro que se moverá cuando el usuario toque la pantalla y arrastre el dedo para mover dicho *joystick*. El *script* que controla el seguimiento del *Stick* en referencia al punto marcado por *Center* se llama **JoystickFunction**, el cual es un *script* proporcionado por el *asset*.

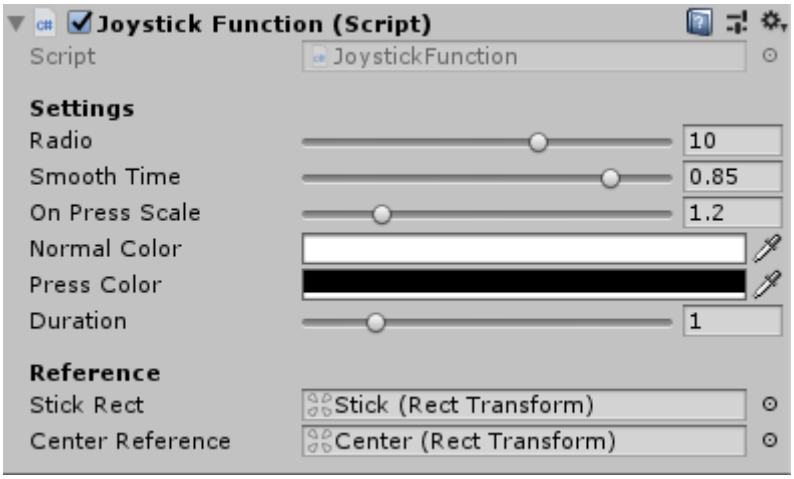

Figura 108. Script Joystick Function

El procedimiento que se ha tenido que programar es el relacionado con el movimiento del avatar y la cámara cuando su *joystick* correspondiente detecte movimiento.

En el caso del movimiento, esto se ha programado dentro del *script* *PlayerCode*, el cual también gestiona el movimiento del avatar en la versión de PC. En la Figura 109 se muestra parte del código que realiza esta función:

```
if (!lose)
{
    //Movement
    float v = JoystickMovement.Vertical / 4;
    float h = JoystickMovement.Horizontal / 4;


    moveDirection = new Vector3(h, 0.0f, v);
    moveDirection = transform.TransformDirection(moveDirection);
    moveDirection = moveDirection * speed;
    controller.Move(moveDirection * Time.deltaTime);
}
```

Figura 109. Código para mover el avatar con el joystick

El movimiento se realiza con el mismo componente que el usado en la versión de PC (este el componente *Character Controller*). En el código se puede ver como las variables “v” y “h” recogen el movimiento del *joystick* (a través de una función que proporciona el *asset*) y estas variables se traducen en un movimiento del avatar a una velocidad igual que la de la versión de PC. Además, también se puede ver como este código se encuentra contenido dentro de una sentencia *if* que comprueba que el jugador solo es capaz de moverse únicamente cuando no haya perdido la partida. Esto evita que un jugador derrotado sea capaz de moverse y entorpecer el normal funcionamiento de la partida.

La gestión del movimiento de la cámara cuando el *Stick* de su *joystick* correspondiente detecta movimiento se realiza de forma independiente. El *script* se llama ***Controller Camera Joystick*** y está añadido al *GameObject* del jugador.

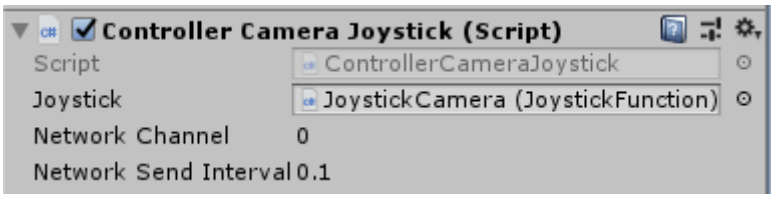

Figura 110. Script Controller Camera Joystick

La función de este script es básicamente traducir el movimiento detectado en el *joystick* izquierdo en un movimiento de la cámara del jugador. En la Figura 110 se muestra la porción del código de *Controller Camera Joystick* que realiza esta tarea.

```
if (!isAndroid)
    return;

    float v = Joystick.Vertical;
    float h = Joystick.Horizontal;

    vertical += v/4;
    horizontal += h/4;

    controller.transform.eulerAngles = new Vector3(-vertical, horizontal, 0.0f);
```

Figura 111. Código para el movimiento de la cámara con el joystick

Como se puede ver, hay mucha similitud con el fragmento de código representado en la Figura 111. El movimiento detectado se guarda en las variables “v” y “h”, a su vez, se divide entre 4, y se guarda dentro de otra nueva variable. Esta división se hace para que el movimiento de la cámara del jugador no sea tan agresivo como el movimiento del *joystick*. Finalmente, en la última se aplica ese movimiento a la cámara del jugador.

6.1.2 Botones y lista de streaming

Para aplicar funcionalidades a los botones se debe realizar lo mismo que con la versión de PC. Se debe incluir el componente ***Button*** y añadir un comportamiento cuando este sea presionado. En la Figura 112, se representa el componente Button añadido al *GameObject* del botón de inicio del streaming (F1). En la sección ***OnClick*** se debe elegir la función que se ejecutará cuando el botón sea presionado. En este caso, la función que se

ejecutará se llama *InitStreaming*, la cual se encuentra dentro del *script CallApp*, estudiando en el apartado anterior.

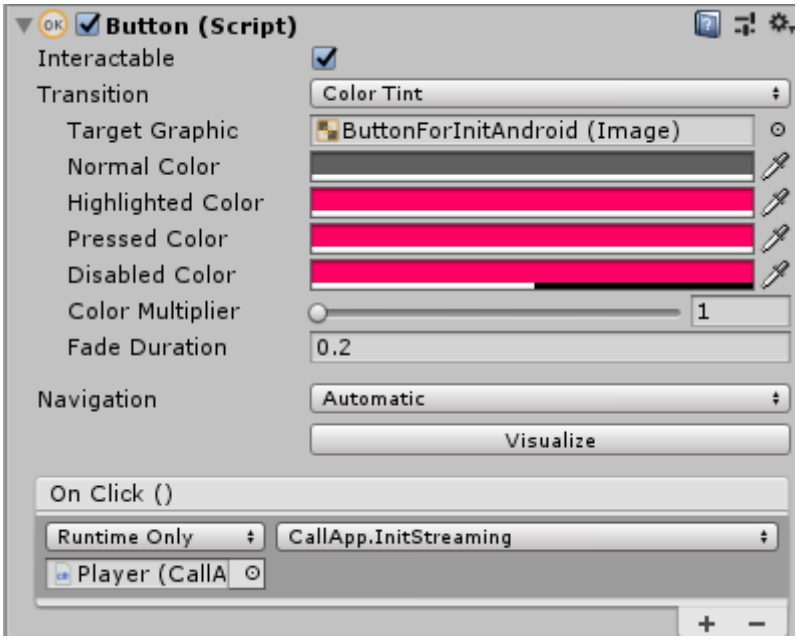

Figura 112. Componente Button para iniciar sala

En el caso de la lista de *streaming activos*, como ya se vio anteriormente en la Figura 105, cuando el jugador presione el área comprendida entre el cuadrado naranja, aparecerá dicha lista. En la Figura 113 se representa la sección de código que realiza esta función.

```
//Open Streaming
Rect bottomRight = new Rect(Screen.width / 2, Screen.height / 2, Screen.width / 2, Screen.height / 2);

if (Input.touchCount == 2)
{
    var touchPos = Input.GetTouch(0).position;
    if (bottomRight.Contains(touchPos) && !StreamingListAndroid.activeSelf)
    {
        StreamingListAndroid.SetActive(true);
        StreamingListAndroid.GetComponent<StreamingList>().OnActivateSearch();

    } else if(bottomRight.Contains(touchPos) && StreamingListAndroid.activeSelf)
    {
        StreamingListAndroid.SetActive(false);
    }
}
```

Figura 113. Código para abrir la lista de streaming en Android

La línea siguiente al comentario crea un rectángulo invisible en la interfaz de usuario del jugador en la parte derecha de la pantalla (justamente la posición del cuadrado naranja). Posteriormente se hace un *if* que detecta cuando hay

dos toques de pantalla. Si detecta que hay dos toques y la lista de *streaming* está abierta, la cerrará En el caso contrario, si detecta que la lista de *streming* está cerrada, la abrirá.

6.2 Compatibilidad con Oculus Go

Unity3D también permite el uso de las *Oculus Go* para sus juegos. Para su implementación en el juego se ha tenido que crear un proyecto aparte, ya que a diferencia de la versión Android, era imposible incluir la versión de las *Oculus Go* junto con las otras dos.

Aplicando nuevamente lo que se ha desarrollado al principio de este capítulo en relación al cambio de plataforma y la API mínima necesaria que se necesita, se debe realizar un paso más para que Unity3D reconozca que el proyecto va a funcionar en unas *Oculus Go*. Se debe ir nuevamente a ***File,*** buscar ***Build Settings*** y abrir el desplegable ***XR Settings***. Finalmente, se deberá marcar la opción ***Vritual Reality Supported*** y añadir la opción *Oculus* dentro de la lista de ***Virtual Reality SDKs***.

Figura 114. Activar la realidad virtual en Unity3D

El primer paso para empezar a programar la implementación de las *Oculus Go* es instalar el *asset* ***Oculus Integration***. Este asset posee una gran cantidad de escenas ejemplo, *scripts*, *GameObjects* y herramientas para incluir las gafas a cualquier juego desarrollado con Unity3D.

6.2.1 Escena del menú inicial

Dentro de los *GameObjects* que proporciona *Oculus Integration*, se utilizarán dos para el caso de la escena del menú inicial (escena UI).

- ***OVRCameraRig:*** Hará la función de cámara principal, por lo que cámara que había en las versiones de PC y Android tendrán que eliminarse.
- ***OVRGazePointer:*** Realizará la función de puntero de las *Oculus*. Este *GameObject* creará el puntero de selección con el cual podremos seleccionar los distintos modos de creación de partida. El puntero seguirá en todo momento el centro de las gafas. Por lo tanto, no se utilizará el mando de las *Oculus* para moverse, el mismo movimiento de la cabeza será necesario.
- ***EventSystem:*** Se debe colocar para gestionar la interacción con el mando y el menú.

El *GameObject OVRCameraRig* estará comprendido por los siguientes *scripts* que aparecen en la Figura 115.

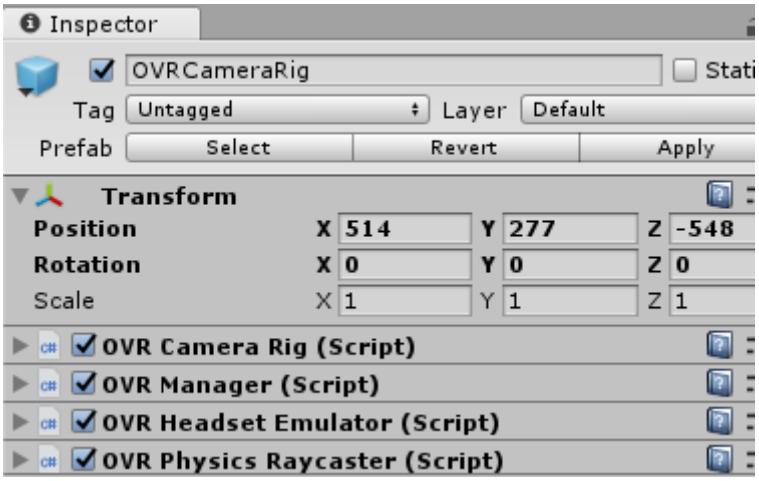

Figura 115. GameObject OVRCameraRig

Los *scripts* los proporciona el *asset*, y cada uno de ellos tienen unos parámetros de entrada, los cuáles se mostrarán a continuación:

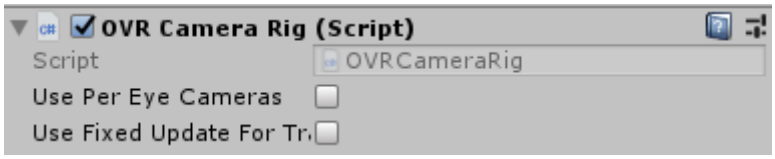

Figura 116. Componente OVR Camera Rig

Figura 117. Componente Headset Emulator

Figura 118. Componente Physics Raycaster

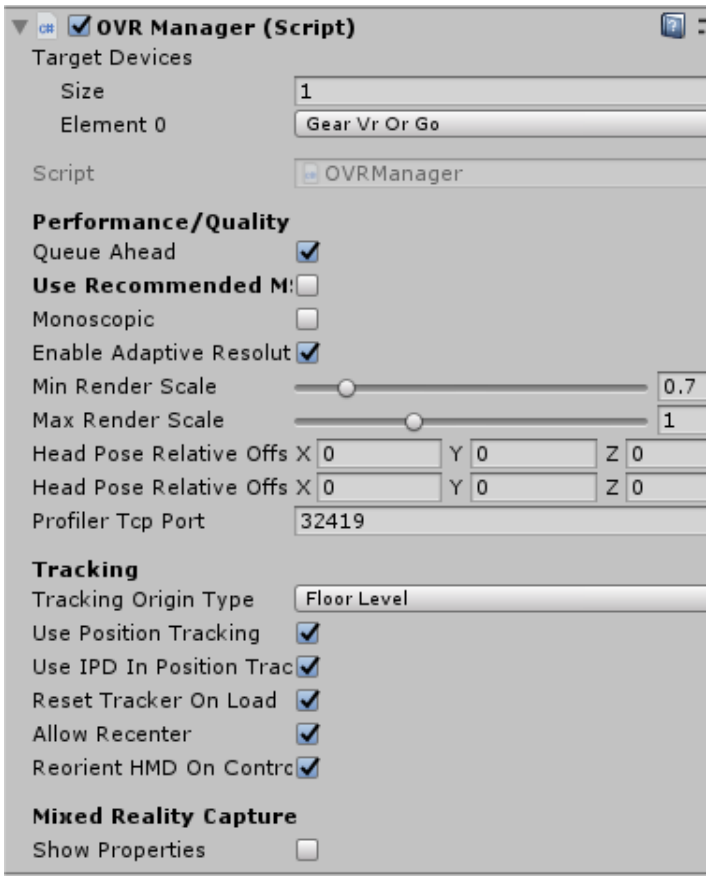

Figura 119. Componente OVR Manager

El *GameObject OVRGazePointer* representará el punto que aparecerá en la pantalla para seleccionar las opciones de creación de partida. Para acceder a las opciones habrá el pulsar el disparar del mando de las *Oculus Go*. La Figura 120 muestra el *GameObject* y su *GameObjects* hijo:

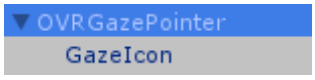

Figura 120. GameObject OVR GazePointer

El *GameObject OVRGazePointer* tiene los siguientes componentes añadidos a él.

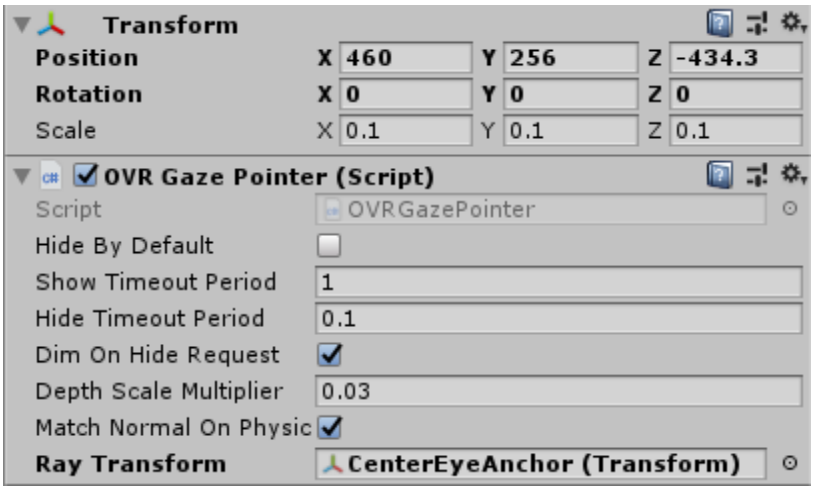

Figura 121. Componentes de OVR Gaze Pointer

El componente *OVRGazePointer* debe tener como parámetro de entrada de ***Ray Transform*** la posición de la cámara central que aparece como componente hijo del *GameObject* llamado *OVRCameraRig* tratado anteriormente. De esta manera, el puntero seguirá al centro de las gafas *Oculus*, de modo que se podrá controlar el puntero girando la cabeza al sitio que se desee.

El componente hijo de *OVRGazePointer* tiene varios componentes:

- *Transform*
- *Mesh Render*
- *Quad*
- *OVR Overlay*

Los 3 componentes primeros han sido muy utilizados a lo largo de este documento. El más importante y el que guarda relación con el funcionamiento de las *Oculus* es el componente ***OVR Overlay*****,** el cuál programa el funcionamiento del cursor y la imagen que tendrá este una vez que se ejecute el juego. En la Figura 122 se puede ver este componente, y, además, se puede ver que textura se ha elegido poner para el puntero.

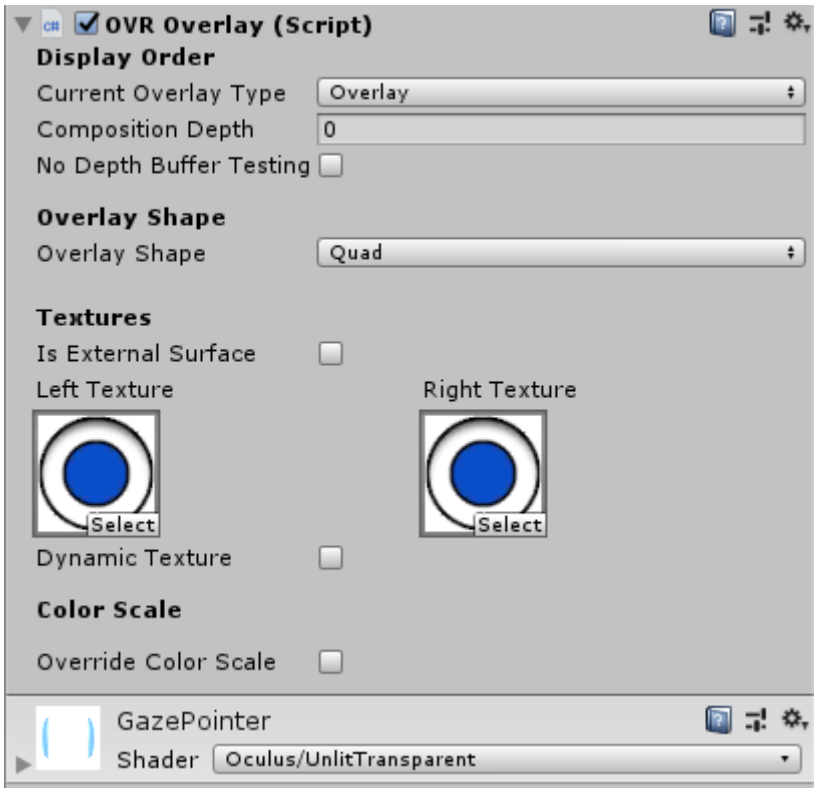

Figura 122. Componente OVR Overlay

6.2.2 Escena Game

Para la escena Game el cambio ha sido más sencillo. El *GameObject MainCamera* de las versiones de PC y Android ha sido eliminado de la escena. En su lugar, se ha utilizado el *GameObject OVRCamera*, se le ha cambiado el nombre a *MainCamera* y se ha colocado como *GameObject* hijo del GameObject del Player.

Figura 123. GameObject Player completo

Este *GameObject* solo difiere del de *OVRCameraRig* en que se le ha añadido un componente llamado ***Camera Follow***. Este *script* lo que hace es ubicar al *GameObject MainCamera* en todo momento justo delante del jugador, de manera que la cámara seguirá en todo momento el movimiento del jugador en primera persona. En la Figura 124 se muestra los componentes que tiene el *GameObject MainCamera*:

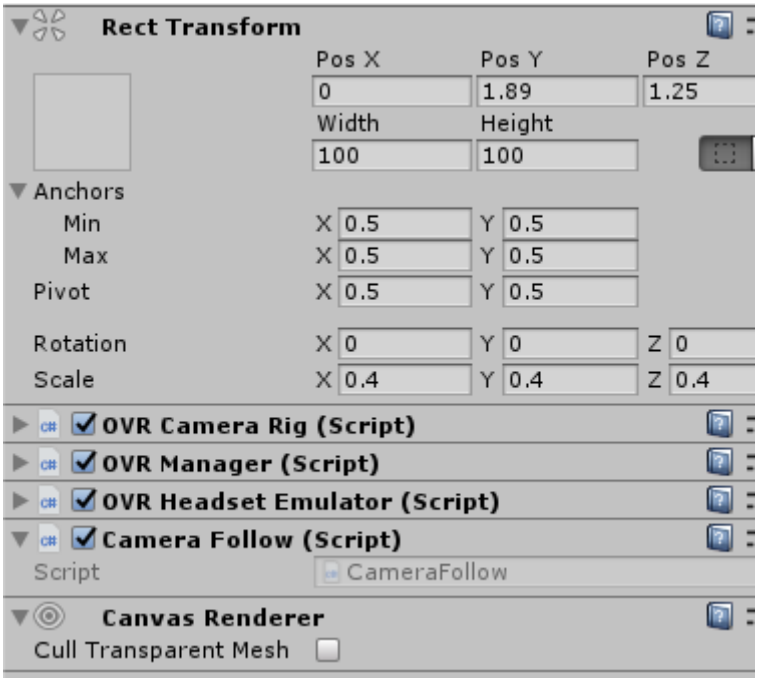

Figura 124. GameObject MainCamera del jugador

Una vez hecho esto, en el script *PlayerCode* del jugador se ha añadido tres líneas para que se recoge el movimiento que se realiza con la cabeza y se traduzca en un movimiento de cámara (y por lo tanto una rotación del avatar) y, además, el avatar se pueda mediante los controles que trae el mando de *Oculus*. En la siguiente Figura 125 se muestran estas líneas de código.

```
109     if (!lose)
110     {
111         //Code for moving avatar
112         moveDirection = OVRInput.Get(OVRInput.Axis2D.PrimaryTouchpad);
113         transform.Translate(Vector3.forward * speed * moveDirection.y * Time.deltaTime);
114         transform.Translate(Vector3.right * speed * moveDirection.x * Time.deltaTime);
115         transform.localEulerAngles = new Vector3(CenterCamera.transform.eulerAngles.x, CenterCamera.transform.eulerAngles.y, 0);
116     }
```

Figura 125. Código para mover la cámara y el avatar

En la Figura 125, la línea 112 recoge el movimiento detectado por el mando de *Oculus*. En las líneas 113 y 114 se produce el movimiento vertical o lateral del jugador. Y por último en la línea 115 se produce la rotación del avatar junto con el movimiento de la cámara.

7. Pruebas de rendimiento de la red

Este apartado se centra en estudiar una serie de pruebas realizas a partir de posibles condiciones de stress (algunos de ellas atípicos) para estudiar el comportamiento de la API de *Networking* en situaciones normales y de carga excesiva. La forma de extraer estos datos ha variado a lo largo del tiempo, ya que, por desgracia, al enviarse los paquetes a través de UDP en vez de sobre RTP en la red no se ha podido utilizar *Wireshark* para calcular automáticamente el retardo o *jitter* que sufren los paquetes que intercambia Unity en los distintos escenarios, en base a las marcas temporales.

En primer lugar, se optó por crear un Log con Unity, el cuál guardara en un archivo de texto el momento exacto el que el jugador dispara una bala. Y, además, otro archivo en el servidor que también señalara el tiempo exacto en el que la posición de la bala es distribuida a todos los clientes. Pero desgraciadamente, los valores obtenidos no tenían sentido en algunas ocasiones debido a la falta de sincronización de los relojes de los sistemas.
Finalmente, se optó por utilizar una herramienta que posee Unity para el control del rendimiento del juego en tiempo a real. Esta herramienta se llama *Profiler.* En la Figura 126 se muestra la interfaz de esta herramienta en ejecución.

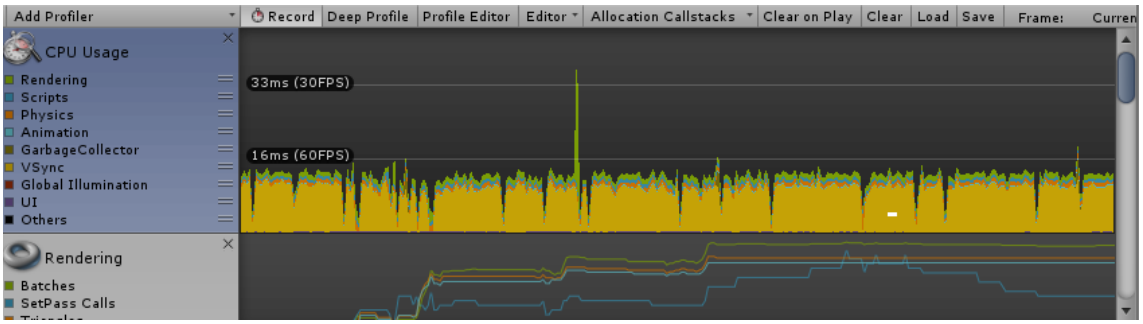

Figura 126. Herramienta Profiler en funcionamiento

Mediante la herramienta de la Figura 126, Unity muestra el retardo que sufre cada script, renderizado, motor de física, etc. Este retardo generalmente es del orden de milisegundos.

De los elementos que Unity estudia en su Profiler, solo se ha extraído en un archivo de texto el valor de ***NetworkIdentity.UNetStaticUpdate.*** Este script interno de Unity se encarga de renderizar y redistribuir la escena del servidor al resto de los clientes. De modo, que cuanto mayor carga deba soportar este script, a menor velocidad se ejecutará. Lo normal es que su valor más grande ocurra justo al principio de iniciar la partida, ya que es en este momento cuando el servidor debe redistribuir todos los elementos iniciales del mapa a los clientes.

El estudio se llevará a cabo en dos escenarios clave, que permita analizar el efecto de la mayor o menor limitación del ancho de banda disponible; por medio de su ejecución en una red local cableada y en una red local inalámbrica. Como es de esperar, se espera un mejor rendimiento en la red cableada, pero el objetivo es el de delimitar el efecto negativo que el uso de otras redes típicas en entornos domésticos y laborales puede tener. La red cableada es la del Laboratorio de Simulación de la Universidad Politécnica de Linares. A su vez, la red inalámbrica empleada para las pruebas ha sido la red de *telema2*, también parte de las infraestructuras del departamento de Ingeniería de Telecomunicación.
En cada escenario (ya sea cableado o inalámbrico) se analizarán los siguientes tres casos de uso:

- **Caso 1:** Se estudiará la tendencia de subida que hay en la carga de ***UNetStaticUpdate*** cuando progresivamente se le van añadiendo jugadores a cada partida.

- **Caso 2:** Se estudiará como sufre la red cuando hay un jugador que se comporta de manera maliciosa dentro de la red. Este jugador ha sido programado para que dispare por cada actualización de la función *Update.*

- **Caso 3:** En este apartado se estudiará el caso de incluir conferencias entre 2, 4 hasta 6 jugadores para ver como carga esto la red.

Se han programado dos tipos de jugadores que serán de ayuda en la realización de estas pruebas.

- **Jugador IA:** son aquellos que se han programado en el juego para que se comporten de forma autónoma y tengan cierta inteligencia artificial. De esta manera se emula que este jugador está siendo controlado por un humano y simula su comportamiento en la red.
- **Jugador malicioso (JM):** son aquellos que han sido programados para que disparen por cada vez que se ejecute la función *Update* de Unity3D. Esta función permite el disparo de 51 balas por segundo.
- **Un jugador normal (J):** será un jugador controlado por una persona que intenta ser el vencedor.

Los datos que se van a extraer para cada caso corresponden al valor en **milisegundos** de **NetworkIdentity.UNetStaticUpdate** obtenido por el Profiler en cada trama. Este valor indica cuánto tiempo tarda la escena en sincronizar todos los *GameObjects* de todos los elementos que tienen relación con la API de Networking. Por lo tanto, cuanto mayor es el número de milisegundos obtenidos, mayor será la carga en la red debido a una mayor lentitud en la actualización. Para cada caso se recogerán un total de **1000 muestras** para que los resultados sean más precisos. Con estas muestras se procederá a hacer la **mediana,** ya que no se asegura que la distribución sea normal, debido a la existencia de posibles valores muy dispares causados por un mal funcionamiento de la red o del propio juego. Lo más correcto es utilizar la mediana para poder ignorar en cierta medida los datos atípicos, y poder obtener más fiablemente un valor que represente el funcionamiento **normal** de la red. Por ejemplo, imaginemos que tenemos 1000 muestras cuyos valores oscilan entre 0,1 y 0,4 ms (como en el caso 1.1). Si dentro de estas muestras aparece un valor atípico (imaginemos que de 20 ms), este valor aumentará absurdamente la media (en torno a 0.2). Además, este valor atípico se debe seguramente a algún mal funcionamiento de la red por algún motivo ajeno al juego, o bien por algún fallo puntual de la CPU. Por lo tanto, no estaríamos extrayendo un dato que nos diera el **tiempo normal** que tarda

Unity en sincronizar la escena. Con la mediana, se obtiene el valor central de los datos, y se evita que los datos atípicos intervengan negativamente a no ser que estos sean mayoría.

7.1 Red cableada como escenario de pruebas

Los PCs de este laboratorio se corresponden con equipos de: procesador i5 de 3.2Ghz, 8Gb de RAM, 250Gb de SSD y una tarjeta gráfica Nvidia Geforce 210. El servidor del juego ha sido también uno de estos equipos del laboratorio. Las muestras se extraen justo después del inicio de la partida en todos los equipos. En el caso de las características de la red, son las siguientes: tarjetas Realtek PCI Express Gigabyte Ethernet utilizando cables de categoría 6.

7.1.1 Caso 1: Incremento paulatino de jugadores

Se empezará a aumentar progresivamente de 2 en 2 el número de jugadores en cada partida.

Casos	Escenarios	Medianas
Caso 1.1	2 jugadores (1 IA y 1 jugador	0.144 ms
Caso 1.2	4 jugadores (3 IA y 1 jugador)	0.174 ms
Caso 1.3	6 jugadores (5 IA y 1 jugador)	0.189 ms
Caso 1.4	8 jugadores (7 IA y 1 jugador)	0.244 ms
Caso 1.5	10 jugadores (9 IA y 1 jugador)	0.287 ms

Tabla 1. Tabla del caso 1 en la red cableada

Se puede apreciar como hay una tendencia ascendente de unos milisegundos de más por cada jugador que se añade a la partida, aunque esté lejos de alcanzar el número máximo ya que los valores no varían significativamente. Este incremento simboliza que por cada jugador nuevo en la partida se incrementa un **0,3ms** el tiempo que tarda el servidor en sincronizar la escena en todos los clientes.

7.1.2 Caso 2: Presencia de jugadores maliciosos

Se empezará a aumentar progresivamente el número de **jugadores maliciosos** de 1 en 1 cada partida. Los jugadores maliciosos cargarán la red notablemente. Este estudio permitirá llevar al límite las capacidades de la API para comprobar cuánto es capaz de aguantar.

Casos	Escenarios	Medianas
Caso 2.1	10 jugadores (1 jugador, 8 IA y 1 JM)	0.602 ms
Caso 2.2	10 jugadores (1 jugador, 7 IA y 2 JM)	0.931 ms
Caso 2.3	10 jugadores (1 jugador, 6 IA y 3 JM)	1.338 ms
Caso 2.4	10 jugadores (1 jugador, 5 IA y 4 JM)	2.974 ms
Caso 2.5	10 jugadores (1 jugador, 4 IA y 5 JM)	3.252 ms

Tabla 2. Tabla del caso 2 en la red cableada

Se puede apreciar como hay una tendencia ascendente de unos milisegundos de más por cada jugador malicioso que se añade a la partida, Este incremento es notable, y va aumentando por cada jugador malicioso que se añade. Esto demuestra que este tipo de jugadores son altamente perjudiciales para el juego.

En el caso 2.5 ya se pueden apreciar valores muy atípicos (de hasta 35ms). Aquí la API de Unity3D está empezando a fallar y empieza a mostrar mensajes del tipo que se muestra en la Figura 127.

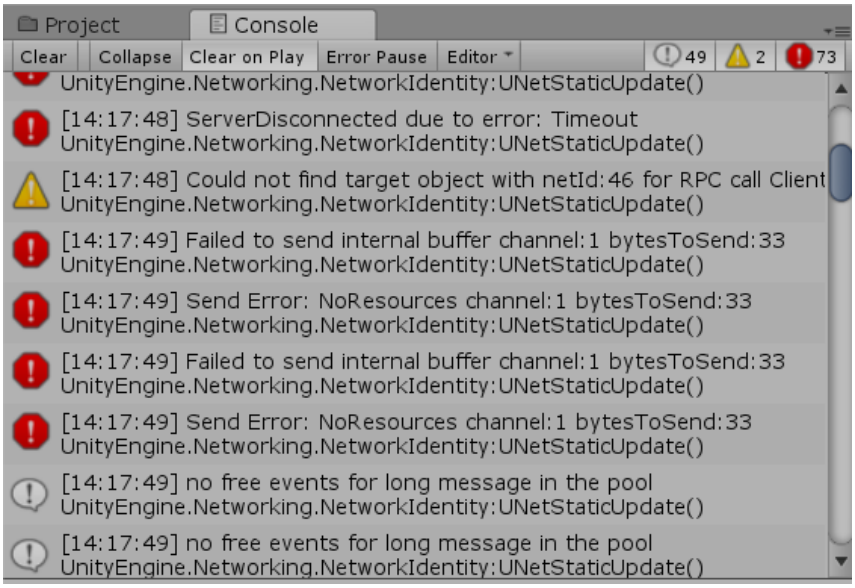

Figura 127. Fallo en la red

El número máximo de jugadores maliciosos que el juego es capaz de soportar es de 4, a partir de este número, el juego será inviable ya que aparecerá fallos que repercuten en las funciones básicas, como las colisiones. Además, podemos ver que cuando la mediana de los datos llega a **3ms**, empieza a haber problemas, dado que el efecto negativo del envío masivo de datos en la red afecta al propio renderizado y sincronización del juego en los distintos equipos. En la Figura 128 se muestra un ejemplo, en donde se puede ver como las balas rebotan cuando estas colisionan con un elemento. Esto no debería suceder, ya que cuando las balas colisiones estas desaparecen de la escena, pero debido a la inmensa carga de datos que hay en ese momento, Unity ni siquiera gestiona correctamente el tema de las colisiones ya que el servidor no es capaz de mandar tantas actualizaciones a los clientes.

Figura 128. Efectos del fallo en la red en el juego

7.1.3 Caso 3: Uso de WebRTC

En este apartado se estudiará cómo afecta el streaming a la red partiendo del caso 2.4 del apartado anterior. Se ha elegido este caso ya que es el caso límite a partir del cual empieza a fallar la red.

Casos	Escenarios	Medianas
Caso 3.1	10 jugadores (2 jugadores en streaming, 4 JM y 4 IA)	2.144 ms
Caso 3.2	10 jugadores (4 jugadores en streaming, 4 JM, 2 IA)	2.804 ms
Caso 3.3	10 jugadores (6 jugadores en streaming, 4JM, 2 IA)	4.120 ms

Tabla 3. Tabla del caso 3 en la red cableada

En el caso 3.3 la API empieza a fallar, y es más apreciable que el número de FPS que se muestran de las cámaras empiezan a decaer considerablemente, además de eso, el sonido empieza a retardarse excesivamente. De estos datos se puede extraer que en el caso 2.5 del apartado anterior la mediana es de 3.25ms, y que en este caso 3.3 la mediana es 4.12ms. En ambos casos, empiezan a aparecer mensajes de que Unity3D falla al enviar los datos. Por lo tanto, podemos hacer una estimación de que cuando ***UNetStaticUpdate*** en el Profiler de Unity3D tiene un retraso de entre 3.25ms aproximadamente, la red comienza a fallar.

7.2 Red inalámbrica como escenario de pruebas

La red inalámbrica empleada ha sido la de ***telema2***, y como se podrá prever los resultados obtenidos dejan en evidencia que la red inalámbrica proporciona menos capacidad de envío de datos, debido a las limitaciones de ancho de banda y a las características de soporte de los protocolos de las capas inferiores que dan soporte a la transmisión en un medio inalámbrico, que la red cableada. Los equipos utilizados han sido los mismos que los usados en los casos de la red cableada. Los equipos utilizan tarjetas TP Link 300 Mb/s Wireless y se encuentran conectados a la red por el canal 7.

7.2.1 Caso 1: Incremento paulatino de jugadores

Casos	Escenarios	Medianas
Caso 1.1	2 jugadores (1 IA y 1 jugador	0.156 ms
Caso 1.2	4 jugadores (3 IA y 1 jugador)	0.177 ms
Caso 1.3	6 jugadores (5 IA y 1 jugador)	0.199 ms
Caso 1.4	8 jugadores (7 IA y 1 jugador)	0.249 ms
Caso 1.5	10 jugadores (9 IA y 1 jugador)	0.297 ms

Tabla 4. Tabla del caso 1 en la red inalámbrica

7.2.2 Caso 2: Presencia de jugadores maliciosos

Casos	Escenarios	Medianas
Caso 2.1	10 jugadores (1 jugador, 8 IA y 1 JM)	0.657 ms
Caso 2.2	10 jugadores (1 jugador, 7 IA y 2 JM)	0.986 ms
Caso 2.3	10 jugadores (1 jugador, 6 IA y 3 JM)	1.430 ms
Caso 2.4	10 jugadores (1 jugador, 5 IA y 4 JM)	3.097 ms
Caso 2.5	10 jugadores (1 jugador, 4 IA y 5 JM)	3.576 ms

Tabla 5. Tabla del caso 2 en la red inalámbrica

A partir del **caso 2.4** empiezan a aparecer los mismos errores que la Figura 127. Por lo tanto, estamos ante el caso límite.

7.2.3 Caso 3: Uso de WebRTC

El caso 3 ha sido especial para la red inalámbrica. Se ha intentado abrir salas de *streaming* desde el caso límite tal y como se hizo en el caso 3 con la red cableada. Pero en este caso, incluso con una sola sala de *streaming* abierta aparece el error de la Figura 129.

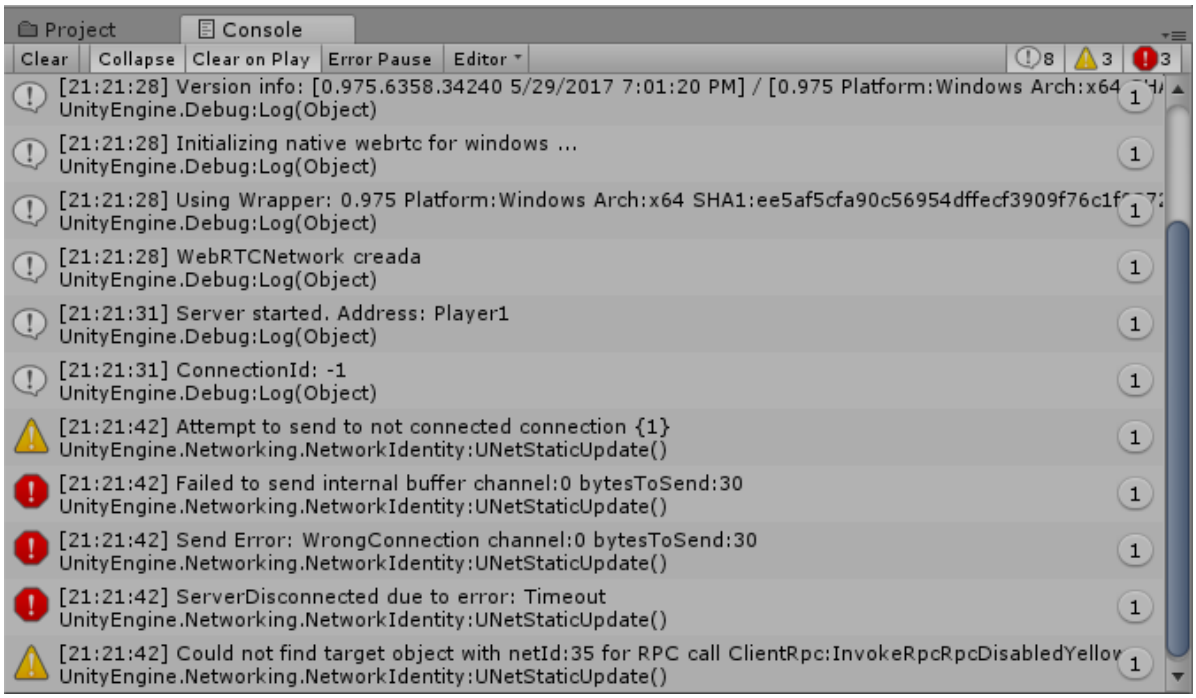

Figura 129. Error al abrir sala de streaming

Casos	Escenarios	Medianas
Caso 3.1	10 jugadores (2 jugadores en streaming, 4 JM y 4 IA)	3.142
Caso 3.2	10 jugadores (4 jugadores en streaming, 4 JM, 2 IA)	3.504
Caso 3.3	10 jugadores (6 jugadores en streaming, 4JM, 2 IA)	4.620

Tabla 6. Tabla del caso 3 en la red inalámbrica

Esto quizás se deba a que Unity tarda 3ms en sincronizar la escena, y siempre que tarda ese tiempo empiezan a aparecer errores, porque no es capaz de gestionar el renderizado completo junto con la ejecución de la funcionalidad asociada a los eventos que se han producido (funciones RPC). Por lo tanto, utilizando la conexión inalámbrica del laboratorio solo se ha podido abrir como máximo 1 sala de *streaming* (caso 3.1), a partir de la segunda sala aparecerán fallos considerables.

7.3 Diferencias entre la red cableada y la inalámbrica

Ya que se han obtenido datos para cada tipo de escenario, ahora sería interesante representar en gráficas la diferencia que hay entre los valores obtenidos para cada caso de un escenario a otro. Se mostrarán a continuación unas gráficas donde se verá el contraste de las medianas de los escenarios inalámbricos y cableados para todos los casos.

Figura 130. Análisis del impacto en los tiempos de ejecución del incremento de jugadores en los escenarios de transmisión (Caso 1.1 y Caso 2.1)

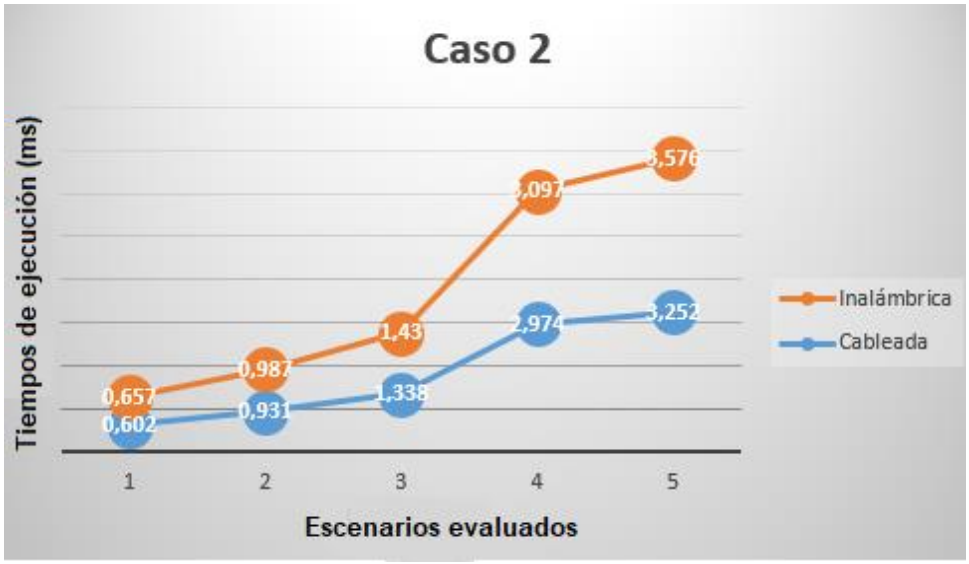

Figura 131. Análisis del impacto en los tiempos de ejecución del incremento de jugadores maliciosos en los escenarios de transmisión (Caso 1.2 y Caso 2.2)

Figura 132. Análisis del impacto en los tiempos de ejecución del incremento de salas de streaming en los escenarios de transmisión (Caso 1.3 y Caso 2.3)

Podemos concluir que Unity tendrá dificultades a la hora de sincronizar una partida que se esté jugando mediante una conexión inalámbrica. Aunque se debe reconocer que los valores no difieren mucho en algunos casos. Además, muestra un buen comportamiento en la parte asociada exclusivamente con la dinámica del juego dado que las limitaciones de ancho de banda son gestionadas de tal manera que la tendencia se mantiene, líneas casi paralelas.

8. Conclusiones

En este proyecto se han cumplido los objetivos que se propusieron al inicio del mismo. En primer lugar, se ha desarrollado un juego en Unity3D utilizando las funciones más importantes que este programa proporciona a los desarrolladores. Posteriormente se le han añadido la posibilidad al juego de ser un título multijugador utiliza la ***API de Networking***.

Además, se ha un estudio de rendimiento e impacto en la red que esta API ocasiona en determinados escenarios y se ha visto la diferencia entre una red cableada y otra inalámbrica. En este aspecto, se puede afirmar que Unity3D ofrece buenas prestaciones en términos de red para el desarrollo de un juego básico multijugador para cualquier plataforma, aunque tampoco se puede considerar una buena opción para desarrollar juegos potentes y comerciales ya que presenta bastantes limitaciones que no son asumibles para juegos que pueden llegar incluso a miles de usuarios al mismo tiempo.

Como siguiente paso, se ha implementado la funcionalidad de una comunicación de datos en tiempo real multimedia o streaming entre dos jugadores. Unity3D carece de componentes o herramientas que permitan realizar esta función, pero cuenta con una buena plataforma llamada *Asset Store* en el que se pueden obtener multitud de herramientas de manera gratuita o de pago, utilizando un framework ampliamente extendido como WebRTC para el intercambio de datos multimedia peer to peer

Finalmente, se ha implementado la compatibilidad con las ***Oculus Go*** e incluso se ha investigado el uso de incorporar la ***Kinect*** al juego**.** Aunque la *Kinect* no mejoraba e incluso complicaba el tipo de gestos necesarios para la interfaz de usuario, para este tipo de juegos. De modo que se puede decir que *Unity3D* posee una muy buena compatibilidad y soporte con la mayoría de periféricos actuales en el mercado.

8.1 Líneas futuras

Tras la finalización del proyecto han surgido nuevas líneas de investigación y mejoras que se podrían implementar. A continuación, se presentan algunas de ellas:

- La creación de un protocolo propio para solucionar la problemática de una llamada multipunto, analizando entre otras estrategias de difusión a nivel de aplicación (ALM-*Application Layer Multicast*).
- Mejora en el diseño gráfico del menú.
- Implementación de una interfaz preparada para unas gafas de Realidad Virtual con 6 grados de libertad, como las *Oculus Rift*.
- Corrección de los posibles fallos que pueda tener el juego en su jugabilidad.
- Mejorar la cámara en la versión de las *Oculus Go* y permitir la escritura de texto para introducir una dirección IP deseada a la hora de unirse a una partida.
- Probar la plataforma *SpatialOS* para mejora la potencia del juego en su versión multijugador.
- Análisis de las prestaciones de alternativas ofrecidas por el uso de servidores dedicados como los implementados por sistemas como los de DarkRiftNetworking.

9. Índice de Figuras

10. Índice de tablas

11. Bibliografía

[1]-Wikipedia- Motor de videojuego -Historia- (Consultado en mayo de 2019)
https://es.wikipedia.org/wiki/Motor_de_videojuego

[2]-Wikipedia – Motores de videojuegos en orden alfabéticos (Consultado en mayo de 2019)
https://es.wikipedia.org/wiki/Anexo:Motores_de_juego

[3]-Wikipedia – Unity3D (motor de videojuego) (Consultado en mayo de 2019)
https://es.wikipedia.org/wiki/Unity3D_(motor_de_juego)

[4]-Wikipedia – Unreal Engine (motor de videojuego) (Consultado en mayo de 2019)
https://es.wikipedia.org/wiki/Unreal_Engine

[5]-Wikipedia – Game Maker Studio (Consultado en junio de 2019)
https://es.wikipedia.org/wiki/GameMaker_Studio

[6]-Fernando Bevilacqua- Building a Peer-to-Peer Multiplayer Networked Game – 12 Aug 2013 (Consultado en junio de 2019)
https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

[7]-Challenges in Peer-to-Peer Gaming (Consultado en junio de 2019)
http://ccr.sigcomm.org/online/files/p2p_gaming.pdf

[8]-Allegro-Wiki- Using XML RPC for your MMORPG (Consultado en junio de 2019)
https://wiki.allegro.cc/index.php?title=Using_XML_RPC_for_your_MMORPG

[9]-Xml-RPC.com (consultado en junio de 2019)
http://xmlrpc.scripting.com/

[10]-IBM- Stream Control Transmission Protocol (Consultado en junio de 2019)
https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.networkcomm/sctp_intro.htm

[11]-Allie Mellen - Explaining the Secure Real-time Transport Protocol (SRTP) May 16, 2018 (Consultado en junio de 2019)
https://www.callstats.io/blog/2018/05/16/a-explaining-the-secure-real-time-transport-protocol-srtp

[12]-Don Glover- UNet Deprecation FAQ- 06 de agosto de 2018 (Consultado en junio de 2019)
https://support.Unity3D.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ

[13]-Brandi House- Multiplayer Connected Games: First steps forward - 12 de septiembre de 2018 (Consultado en junio de 2019)
https://blogs.Unity3D.com/es/2018/09/12/multiplayer-connected-games-first-steps-forward/

[14]-Jeff Grubb- SpatialOS platform for building games of 'unprecedented size' gets open beta - 27 de febrero de 2017 (Consultado en junio de 2019)
https://venturebeat.com/2017/02/27/spatialos-platform-for-building-games-of-unprecedented-size-gets-open-beta/

[15]-Sam Loveridge - What is Google Stadia? Google's game streaming service explained – 06 de junio de 2019 – (Consultado en junio de 2019)
https://www.gamesradar.com/google-stadia-release-date-price-games-controller/

[16]-Sam Dutton- WebRTC in the real world: STUN, TURN and signaling– 04 de noviembre de 2013 – (Consultado en junio de 2019)
https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

[17]-Manual Unity3D – GameObjects - 2018
https://docs.Unity3D.com/es/current/Manual/GameObjects.html

[18]-Manual Unity3D – Componentes- 2018
https://docs.Unity3D.com/es/current/Manual/UsingComponents.html

[19]-Stream Control Transmission Protocol (SCTP)-Based Media Transport in the Session Description Protocol (SDP) – 30 de junio de 2013 (Consultado en junio de 2019)
https://tools.ietf.org/html/draft-ietf-mmusic-sctp-sdp-04

12. Anexos

12.1 Servidor de señalización local

A pesar de que el proyecto utilice un servidor de señalización público aportado por el asset de WebRTC Network, también se provee de un servidor de señalización el cual ha sido adaptado para que funcione acorde a las necesidades del proyecto. Por lo tanto, no se debe utilizar el servidor que proporciona el asset, puesto que este no está adaptado al juego desarrollado en este proyecto.

Lo primero que se necesita para poder hacer funcionar el servidor es instalar algún IDE. En este proyecto se ha utilizado Eclipse. Posteriormente se debe instalar Node.js. Esto se puede hacer desde la página oficial: https://nodejs.org/es/download/.

En función del sistema operativo que se utilice la instalación será de una forma u otra. El proyecto ha sido desarrollado en Windows 10, por lo tanto, en nuestro caso solo ha sido necesario descargar el instalar correspondiente y ejecutarlo.
Finalmente, dentro del Marketplace de Eclipse se ha descargado los siguientes paquetes representados en la Figura 134.

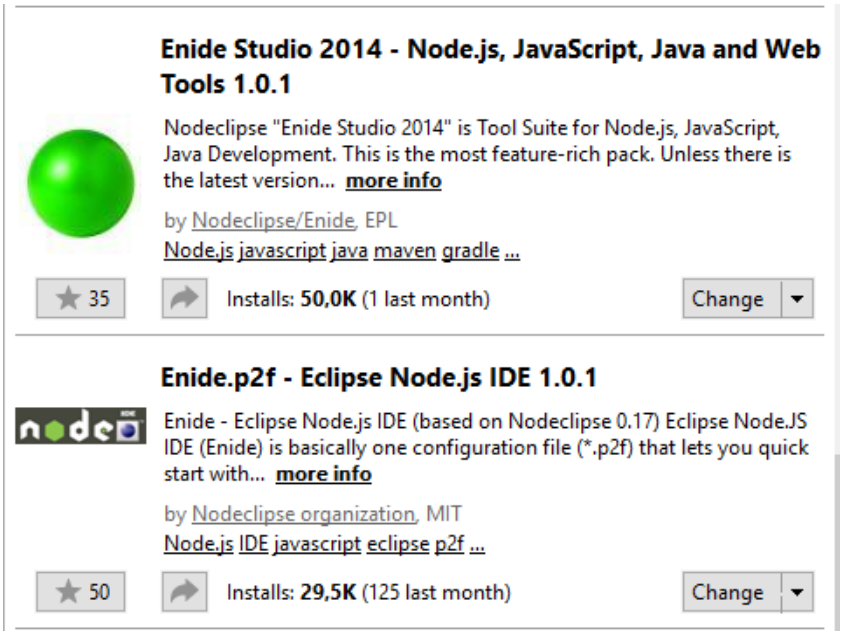

Figura 133. Paquetes instalados en Eclipse

12.2 Instalación de la SDK de Android para Unity3D

Uno de los pasos primordiales para que *Unity3D* pueda compilar sus proyectos para el sistema operativo Android (necesario para la versión de móvil y las *Oculus Go*) es indicar la ruta del *SDK* de Android. En primer lugar, se debe instalar el programa ***Android Studio*** desde la página oficial: https://developer.android.com/studio

Se debe descargar la última versión que haya, y basta con instalar lo que venga por defecto en el instalador de *Android Studio*. A continuación, abriremos *Android Studio* y buscaremos la opción *SDK Manager* dentro de ***Configure***. En la Figura 135 se muestra la ubicación de esta opción.

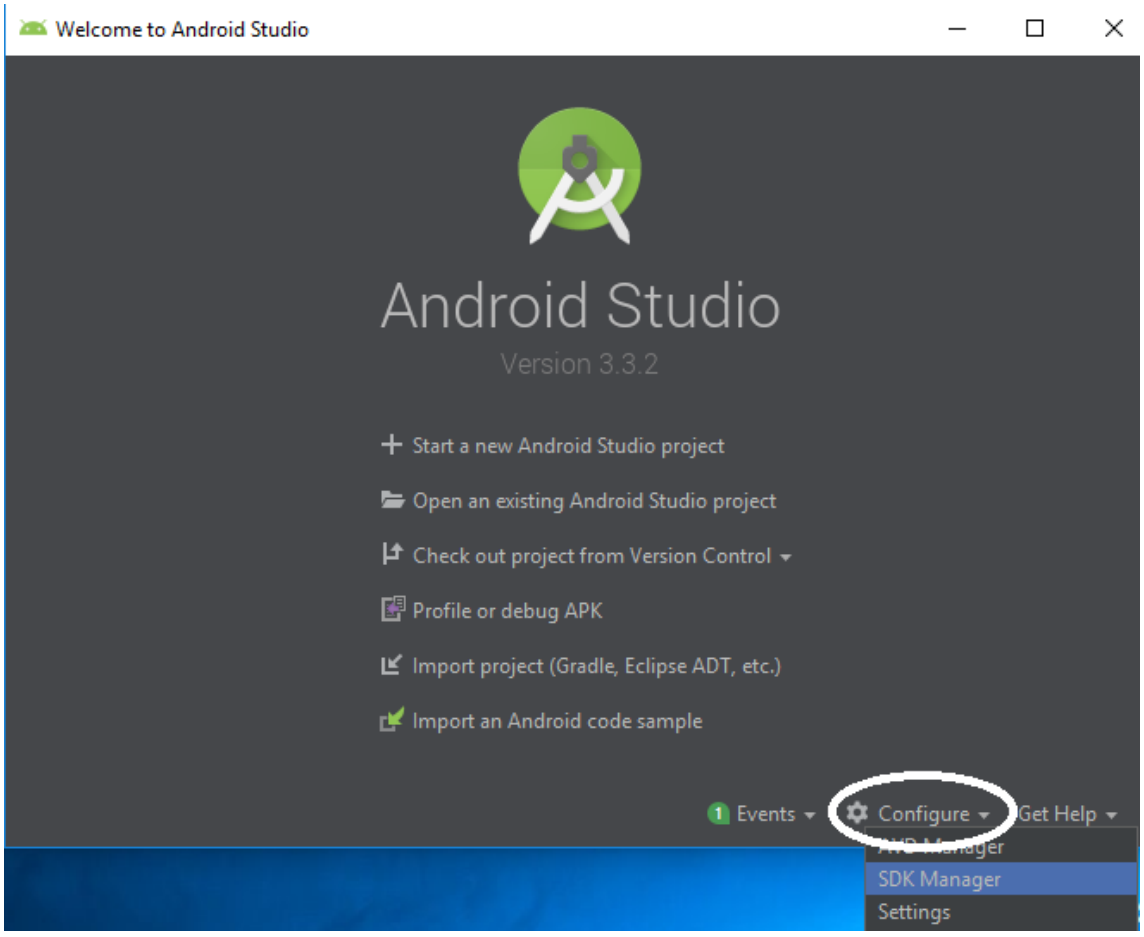

Figura 134. Pantalla inicial de Android Studio

A continuación, se abrirá una ventana donde habrá que copiar y copiamos la ruta que aparece en la parte superior de dicha ventana.

Figura 135. Ruta de los SDK

Una vez copiada la ruta, abriremos Unity3D y se hará click en la opción Preferences, la cual se encuentra dentro de la pestaña Edit de la parte superior de la pantalla. Cuando se abra una ventana, habrá que ir la pestaña External Tools y pegar la ruta anteriormente copiada en la opción SDK. En la Figura 137 se señala el lugar donde hay que pegarla.

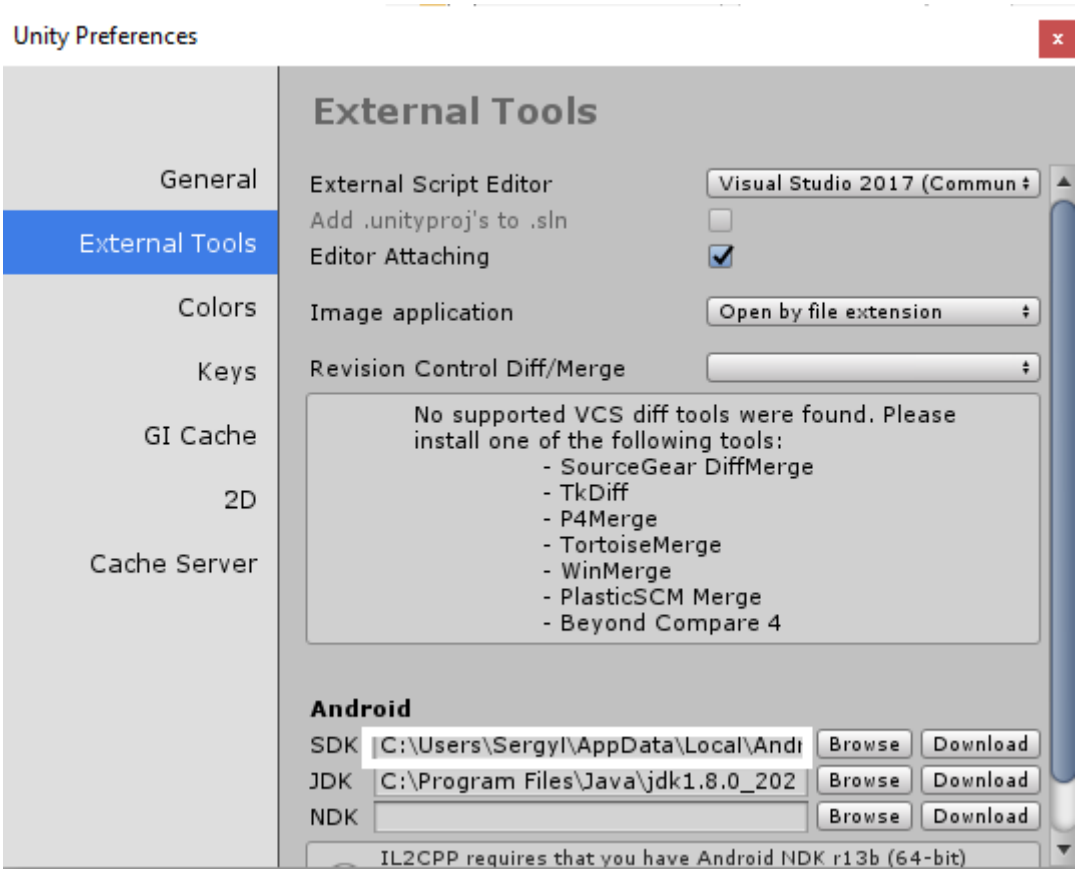

Figura 136. SDK en Unity3D

Una vez hecho esto, Unity3D podrá compilar y generar un .apk para proyectos que estén destinados al sistema operativo Android.