UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de un videojuego multijugador por oleadas en Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Juan Pérez, Jaime

Tutor: Lluch Crespo, Javier

Curso 2017/2018

Resumen

El mercado de los videojuegos está formado por varias plataformas de juego, hecho implica que algunos géneros no se hayan explorado lo suficiente en algunas de ellas. En este proyecto se ha desarrollado un videojuego perteneciente al género horda orientado a la plataforma móvil Android utilizando el motor Unity3D para su creación. Se trata de un juego caracterizado porque los jugadores deben eliminar oleadas de enemigos y superar un número limitado de rondas. Se proporcionan dos modalidades de juego, una en solitario, donde el jugador se enfrenta solo a las oleadas de enemigos y otra multijugador, en la que además interviene otro jugador con el que tendrá que decidir luchar o cooperar.

Palabras clave: Videojuego, Unity, oleadas, multijugador, horda.

Abstract

The videogame market is made up of several gaming platforms, fact that implies that some genres have not been sufficiently explored in some of them. In this project, a horde genre videogame has been developed oriented towards Android mobile platform, using Unity3D engine for its creation. It is a game characterized because players must eliminate waves of enemies and overcome a limited number of rounds. Two play modes are provided, one in solitary, where waves of enemies are faced alone by the player and another multiplayer mode, in which one more player takes part and brings the controversy between cooperation and fight.

Keywords: Videogame, Unity, waves, multiplayer, horde.

Resum

El mercat dels videojocs està format per diverses plataformes de joc, fet que implica que alguns gèneres no s'hagin explorat prou en algunes d'elles. En aquest Project s'ha desenvolupat un videojoc pertanyent al gènere horda orientat a la plataforma mòbil Android utilitzant el motor Unity3D per a la seua creació. Es tracta d'un joc caracteritzat perquè els jugadors han d'eliminar onades d'enemics i superar un numero limitat de rondes. Es proporcionen dos modalitats de joc, una en solitari, on el jugador s'enfronta a soles a les onades d'enemics i una altra multijugador, en la que a més intervé un altre jugador amb qui haurà de decidir lluitar o cooperar

Paraules clau: Videojoc, Unity, onada, multijugador, horda.

Índice de contenidos

Índice de imágenes

Índice de tablas

Definiciones, abreviaturas y acrónimos

AAA

Forma de clasificar a los videojuegos desarrollados por grandes compañías, que han teniendo grandes campañas de *marketing* y un alto presupuesto.

FPS

Siglas correspondientes a *First Person Shooter*. Se trata de una modalidad de juego de disparos en primera persona.

Frame

Fotograma estático que pertenece al conjunto de imágenes que forman una animación.

Trigger

Elemento que es disparado tras cumplirse una serie de condiciones.

Renderizado

Proceso de generar una imagen a partir de un modelo 2D o 3D formado por diversos polígonos.

Corrutina

Función que puede pausar su ejecución durante un tiempo y después retornar su tarea.

Tercera persona

Estilo de cámara dentro del ámbito de los videojuegos que se sitúa detrás del personaje manejado.

Spawn

Punto de aparición en un videojuego.

Lobby

Pantalla característica de los juegos en línea, donde los jugadores se reúnen y esperan el inicio de la partida.

Host

Jugador que ejerce el papel de servidor en una partida.

1. Introducción

Hoy en día, la industria del videojuego es reconocida como una de las más grandes, este título lo ha conseguido gracias a todos los consumidores que reúne y la gran cantidad de dinero que produce, 15.8 millones de usuarios y 1.359 millones de euros en España durante 2017 [1]. Para comprobar estos hechos no hace falta que acudamos a prestigiosos estudios estadísticos, basta con salir a la calle un día y observar cuantos ciudadanos toman cinco minutos de su tiempo para jugar en sus dispositivos.

Entre las opciones que brinda la industria encontramos de todos los tipos y gustos, desde las opciones más tradicionales como pueden ser un ordenador o una videoconsola, hasta los modelos portátiles y los *Smartphone.* Estos últimos han presentado un crecimiento espectacular en los últimos años y su logro se debe principalmente a la posibilidad que ofrecen al usuario de jugar en cualquier sitio [2].

La plataforma elegida para el desarrollo del videojuego, pertenece al mercado de dispositivos móviles y se trata de Android. Actualmente, este sistema operativo es el más utilizado en los *Smartphone* españoles y en 2017 supusieron un 92% de las ventas de este tipo de dispositivos en España [3]. Por otra parte, pese a que la aplicación no va a ser publicada en la tienda de Android, ofrece una serie de facilidades a un coste reducido en comparación con otras plataformas para la publicación en un futuro.

Los videojuegos presentes en la plataforma Android están orientados principalmente a entretener al usuario durante un breve periodo de tiempo. Son muy variados los géneros de juegos que podemos encontrar, pero en concreto nuestro proyecto se centra en los de tipo horda. Este estilo de juegos podemos encontrarlos en otro tipo de plataformas pero no son muy comunes en los dispositivos móviles, hecho que beneficia a la aplicación desarrollada. Los juegos más similares al género horda en los dispositivos Android son los llamados *tower defense,* donde el jugador debe de colocar estructuras que ataquen a los enemigos para defender cierta zona.

El presente proyecto cuyo título se corresponde con “Desarrollo de un videojuego multijugador por oleadas en Unity” tiene por motivación la creación de un primer videojuego y el descubrimiento de la industria como creador de contenido. Siguiendo el género horda se ha creado un juego adaptado para dispositivos móviles el cual presenta dos modos de juego, uno solitario y otro multijugador.

1.1. Objetivos

El principal objetivo de este proyecto es el desarrollo de un videojuego multijugador perteneciente al género horda y orientado para dispositivos móviles Android. El cumplimiento de este objetivo supondrá la posibilidad de disfrutar del juego con otras personas a distancia. Al centrarnos en el género horda, nuestro desarrollo debe de cumplir una serie de requisitos propios del género. Para pertenecer al género es imprescindible que entre las características de nuestro proyecto se incluyan una serie de rondas de enemigos y que el número de rondas por partida sea limitado.

Pese a no considerarse en primer lugar un desarrollo de una modalidad de juego en solitario, esta opción a acabado convirtiéndose en uno de los objetivos del proyecto. Los dispositivos móviles normalmente cuentan con conexión a internet, pero no es tan extraño que se dé el caso en el que el usuario no tiene acceso a la red. Con la creación de este modo se solventa este

problema y permite el disfrute del juego en cualquier situación. Además, esta opción permite abrir un camino a aquellos jugadores que no quieran jugar con otra persona o que quieran practicar antes de enfrentarse a otro jugador.

Finalmente, como último objetivo hemos marcado el diseño del sistema de hordas que todo juego del género debe tener. El diseño del sistema contará con dos enemigos diferentes, uno básico y sencillo, el cual aparecerá más frecuentemente, y otro enemigo más fuerte y difícil de eliminar. El número de rondas debe ser reducido para evitar que las partidas duren demasiado tiempo. Por otra parte, con el paso de las oleadas se debe modificar la dificultad de las rondas y premiar al jugador en función de su avance en una partida.

1.2. Estructura del documento

El documento empieza por la sección 2 dando un repaso a los videojuegos actuales que están relacionados con el modo horda, nombrando las peculiaridades de cada uno y como moldean el género a su juego, prestando especial interés en sus características y cuáles de estas incluirá nuestro proyecto.

Después de realizar una breve presentación, durante el apartado 3, de las herramientas que se han utilizado para desarrollar el proyecto, se hace un recorrido detallado de los aspectos más importantes del motor utilizado para desarrollar el juego, Unity3D. Esto tiene lugar en la sección 4, donde primeramente se expone como se trabaja con el motor, mostrando los elementos que nos proporciona así como la forma de utilizarlos y trabajar con ellos. En la segunda parte de esta sección nos centramos en la explicación de las distintas ventanas, tanto las principales como aquellas con funcionalidad más avanzada.

Seguidamente, se presenta el proceso de diseño del videojuego en la sección 5. Durante este apartado se expone el concepto del juego y la ambientación sobre la que este gira. En esta parte del proyecto se muestran los distintos aspectos que caracterizan al videojuego, como pueden ser la ambientación, los personajes o las interfaces, explicando el diseño que siguen y el porqué de este.

A continuación, en la sección 6 encontramos la implementación del videojuego. Este apartado se encuentra dividido en dos subapartados, el primero trata el modo solitario y el segundo el modo multijugador. A lo largo del primer punto, se explica cómo se han creado las distintas partes que forman el modo solitario. Se repasa en detalle todas las características de las distintas partes, desde los personajes y los componentes que los forman hasta la implementación del sistema de rondas. El punto que trata sobre la modalidad multijugador, parte de la información proporcionada por la explicación del modo solitario. De esta forma, se muestran los cambios y diferencias entre ambas implementaciones obteniendo como resultado final el objetivo principal del proyecto, la creación de un juego multijugador.

Por último, en la sección 7, se exponen las conclusiones obtenidas por la realización del proyecto, repasando los objetivos que se habían marcado y el resultado obtenido. Además de esto, se comentan una serie de mejoras que se podrían implementar en un futuro.

2. Estado del arte

Actualmente la industria del videojuego ofrece un abanico inmenso de posibilidades al consumidor que abarca todo tipo de géneros y gustos, desde un simple juego de móvil hasta un desarrollo AAA. A continuación se repasan una serie de videojuegos relacionados con la temática de nuestro producto y que han resultado de interés para la realización del proyecto.

2.1. Videojuegos con modo horda

Gears of War 4

Gears of War 4 es un videojuego de disparos en tercera persona del género *survival-horror*, desarrollado por la compañía The Coalition y distribuido para sistemas Microsoft.

Este título pertenece a la primera saga en introducir el modo horda en 2008 con Gears of War 2. En este modo de juego, Imagen 1, el jugador tiene que sobrevivir un número determinado de oleadas de enemigos, finalizando la partida tras superarlas todas. Implementa un sistema de puntuación que tiene en cuenta los enemigos eliminados y los supervivientes por ronda, aquellos jugadores eliminados durante una oleada de enemigos son resucitados al comienzo de la siguiente [4].

Imagen 1: Gears of War 4

Call of Duty

Call of duty se trata de una de las mayores sagas de *FPS* de estilo bélico, desarrollada principalmente por Infinity Ward y distribuida por Activision.

Varios de los títulos de esta saga incluyen un modo de juego en el cual se debe sobrevivir a oleadas de enemigos, Imagen 2, y se diferencian de los modos horda en que las rondas son ilimitadas. La acción se sitúa en un mapa cerrado donde con el transcurso de las rondas se van desbloqueando diferentes zonas y armamento. La experiencia se puede disfrutar en solitario o en multijugador y el resultado de las partidas se ve reflejado en un sistema de puntuaciones global [5].

Imagen 2: Modo supervivencia Call of Duty

Killing Floor 2

Killing floor 2 se trata de un videojuego *FPS* desarrollado y publicado por Tripwire Interactive en 2016, el cual es la secuela del primer juego Killing Floor.

El propio videojuego sigue el estilo de los modos horda, donde los jugadores deben de eliminar un número de enemigos que oscila con la cantidad de jugadores. Cuando consiguen esa meta pueden enfrentar al jefe y tras derrotarlo progresan en la historia [6].

Imagen 3: Killing Floor 2

Splatoon 2

Splatoon 2 es un videojuego de disparos en tercera persona, publicado en 2017, con una temática más amigable ya que se utiliza pintura como munición. Ha sido diseñado y desarrollado por Nintendo para la videoconsola Nintendo Switch.

Entre los modos de juego presentes en este título encontramos uno de tipo horda clásico, llamado “salmon run”, Imagen 4, en el que los jugadores deben superar una serie de oleadas. Para superar las rondas, además de derrotar a los enemigos, los jugadores tienen como objetivo la recolección de objetos que generan los enemigos abatidos [7].

Imagen 4: Modo *salmon run* en Splatoon2

Características de los videojuegos

Una vez analizados los diferentes productos que encontramos en el mercado, podemos extraer una serie de características que son de interés para la realización de nuestro proyecto. En la Tabla 1 encontramos las cualidades de los videojuegos mencionados y sus propiedades más interesantes.

Tabla 1: Características de los videojuegos con modalidad horda

Característica	Videojuego			
	Gears of War 4	**Call of Duty**	**Killing Floor 2**	**Splatoon 2**
Número oleadas	Limitado	Ilimitado	Limitado	Limitado
Multijugador	Sí	Sí	Sí	Sí
Modo solitario	Sí	Sí	Sí	No
Incremento dificultad	Sí	Sí	Sí	Sí
Objetivos	No	No	Sí	Sí
Sistema de puntuación	Sí	Sí	No	Sí
Variedad de enemigos	Sí	Sí	Sí	Sí
Elementos interactivos	Sí	Sí	Sí	Sí
Selección de personajes	Sí	No	Sí	No

2.2. Selección de características

Una vez realizado el análisis y obtenidas las características de los distintos productos que se encuentran en el mercado, es hora de establecer las prioridades en el desarrollo de nuestro videojuego. En la Imagen 5 podemos ver el método MoSCoW utilizado para detectar estas propiedades.

Mo

- Modo multijugador
- Número de oleadas limitado
- Sistema de enemigos
  - Incremento de dificultad
  - Diferentes tipos

S

- Sistema de puntuación
- Selección de personajes

Co

- Modo solitario
- Tutorial

W

- Elementos interactivos
- Objetivos
- Varios mapas

Imagen 5: Método MoSCoW

Nuestro videojuego debe de contener tres características que son esenciales para lograr nuestro objetivo. La primera es la creación de un modo multijugador que ofrezca la experiencia de jugar con otra persona. La segunda cualidad a implementar en nuestro producto es el establecimiento de un número limitado de rondas, siendo así un juego puramente de tipo horda, diferenciándose de los de estilo supervivencia en los que no hay límite. Por último, encontramos los enemigos de nuestro videojuego, los cuales deben de ir modificando su dificultad con el transcurso de la partida y tienen que ser de distintos tipos, proporcionando así variedad en este aspecto.

Respecto a las cualidades que debería de implementar nuestro producto podemos hacer una división en dos frentes. La primera parte consiste en la implementación de un sistema de puntuación acorde con el juego, esto implicará realizar una distribución de los puntos en función del desenlace de la partida, repartiéndolos entre los dos jugadores o asignándoselos todos a uno. El otro apartado se corresponde con la creación de al menos dos personajes, con habilidades diferentes e incluso complementarias, para aportar variedad al juego y proporcionar a los jugadores dudas sobre cooperar o luchar con el otro.

En lo referente a las propiedades que se podrían implementar en caso de disponer de más tiempo encontramos dos. El modo multijugador se puede modificar y hacer una adaptación para que un jugador pueda jugar solo, ofreciendo así la posibilidad de jugar cuando no se disponga de acceso a internet. Por otra parte, la creación de un tutorial demostrativo es una opción muy correcta, ya que, pese a que el juego no tendrá gran complejidad, esto facilitará la introducción del juego a usuarios con poca experiencia en el campo de los videojuegos.

Finalmente, encontramos aquellas características que no se implementarán pero que en un futuro se podrían tener en cuenta e introducirlas en alguno de los grupos mencionados anteriormente. Entre estos aspectos encontramos la adición de objetivos durante las partidas, elementos interactivos y la implementación de varios mapas, aspectos que podrían aportar mucho al juego pero que por cuestión de tiempo e inversión creativa no se llevarán a cabo en este momento.

3. Herramientas utilizadas

En el siguiente apartado se introduce el *software* utilizado para trabajar en el proyecto, exponiendo el uso que se le ha dado a cada programa sin entrar en mucho detalle.

Unity3D

Se trata del motor de videojuegos creado por *Unity Technologies*, disponible en varias plataformas de desarrollo, utilizado para el desarrollo de este proyecto. En un apartado posterior profundizaremos en los distintos aspectos del motor, pese a esto, seguidamente se introducen las herramientas más utilizadas del motor.

> **Unity Editor:** entorno de trabajo ofrecido por Unity donde se realiza la mayor parte del desarrollo. Se ha utilizado para construir el videojuego a través de las diferentes opciones que proporciona [8].
>
> **Mecanim:** controlador de animaciones basado en una máquina de estados que nos permite alternar entre animaciones de forma sencilla. Utilizado para dotar de animación a los diferentes personajes y enemigos.

Microsoft Visual Studio

Entorno de desarrollo integrado para sistemas operativos Windows que soporta múltiples lenguajes de programación. Con la instalación del motor viene una versión adaptada que enlaza el programa con el motor, la cual se ha utilizado en este proyecto para realizar la programación en el lenguaje C# [9].

Sublime Text 3

Editor de texto multiplataforma que proporciona diversas funcionalidades. Se ha utilizado de forma complementaria a Microsoft Visual Studio en momentos puntuales [10].

Git

Sistema de control de versiones libre y de código abierto diseñado por Linus Torvals, que permite hacer un seguimiento de los cambios realizados en ficheros y coordinar el trabajo de varias personas sobre estos. Con este sistema se ha realizado la administración y el seguimiento de los ficheros del proyecto [11].

> **GitHub:** se trata de una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git, principalmente utilizada para la creación de programas fuente. En esta plataforma se encuentra el proyecto completo [12].

Audacity

Editor digital de audio gratuito y de código abierto disponible para diversos sistemas operativos. Con él se han realizado las capturas y las ediciones de los diferentes audios que aparecen en el juego [13].

4. Unity3D

En este apartado se introduce el motor utilizado en el desarrollo del proyecto. Con el fin de comprender mejor cómo se ha trabajado se introducirán una serie de conceptos básicos sobre el funcionamiento, las tecnologías y su uso.

4.1. Motor

Unity se trata de un motor multiplataforma accesible para todo el mundo que, pese a tener versiones de pago, las diferencias que encontramos entre estas son pocas y orientadas al ámbito empresarial, permitiendo a un usuario normal disponer de todas las funcionalidades.

Entre las características del motor mencionar el soporte de gráficos 2D y 3D, además de ofrecer funcionalidad en términos de físicas, iluminación, animación y colisiones entre muchos otros. Todo esto puede ser respaldado mediante el uso de herramientas externas que complementen el motor.

A continuación se profundiza en los diversos aspectos del motor cuya comprensión es necesaria para ser consciente del funcionamiento de Unity.

4.1.1. Escenas

Las escenas en Unity son las unidades básicas del proyecto que contienen todos los elementos que van a aparecer durante la ejecución del videojuego. Entre los participantes de una escena podemos encontrar personajes, casas, iluminación o partículas, entre otros [14].

Para comprender el concepto de escena en Unity debemos imaginarnos estas como niveles de un videojuego. Las escenas, al igual que los niveles de un juego, son mostradas una a una y mediante el desplazamiento entre ellas se crea el videojuego. De esta manera, el desarrollador debe unir y comunicar las diferentes escenas para que en cada momento se muestre la adecuada.

Un ejemplo que ilustra el concepto de escena en Unity puede ser el siguiente:

- Tenemos dos escenas, una se corresponde con el menú de nuestro videojuego y otra con el juego. Al jugador se le mostrará primero el menú y cuando seleccione la opción correspondiente para empezar el juego, se realizará un cambio de escenas deshabilitando la primera y activando la segunda.

Finalmente, el proceso de carga de las escenas requiere tiempo, que pese a ser poco, puede llegar a perjudicar la experiencia del usuario en determinados momentos. Es por esto, que la organización de las escenas es importante y la carga es recomendable que se haga de forma asíncrona al juego teniendo así la siguiente escena del juego lista cuando se necesite.

4.1.2. *GameObject*

Un *GameObject* es la unidad básica de trabajo del motor Unity. Todos los elementos que encontramos en una escena son *GameObjects*, estos se pueden crear a través del editor, ya sea desde la jerarquía o desde menú, o mediante código utilizando la clase *GameObject* de Unity [15].

Estos elementos no hacen nada por si solos dado que necesitan una serie de propiedades especiales que los conviertan en lo que el creador desea, ya sea un personaje o un efecto en el juego. Los *GameObjects* son contenedores destinados a guardar distintas piezas que los convertirán en los elementos del juego.

Los elementos que contienen los *GameObjects* se llaman *Components*, estos son las piezas que les proporcionan significado y funcionalidad. Cada objeto puede disponer de diversos *Components* y al igual que los *GameObjects* estos pueden añadirse y eliminarse utilizando el editor o los *scripts* [16].

A continuación se muestran y explican los principales *Components* que nos proporciona Unity y que podemos añadir a los *GameObjects*, como podemos ver en la Imagen 6.

- *Transform:* componente básico que contiene la posición, la escala y la rotación del objeto en el espacio del mundo. Si los objetos tienen relación de parentesco estos valores se calculan en función del padre [17].

- *Rigidbody:* permite que los objetos actúen bajo las leyes de la física. Esto permite que se realicen colisiones más realistas y se diferencian de los *Transform* en que interviene el factor fuerza [18].

- *Scripts:* componente que permite manipular el *GameObjectk* y sus *Components* mediante el uso de código, activando y desactivando estos, además de modificar los distintos parámetros de cada uno. En el siguiente punto profundizaremos un poco más respecto a los *scripts* y la programación.

- *Colliders:* establece un volumen en el cual el objeto puede recibir colisiones. Los hay de distintos tipos destacando principalmente los de tipo *trigger*, que pueden ser atravesados por objetos y mandan una señal cuando esto sucede, y los de tipo sólido que colisionan con otros *colliders*. Todos ellos pueden tener diferentes formas para adaptarse al objeto que pertenecen [19].

- *Animator:* este componente permite dotar de animaciones al objeto mediante el uso de un *Animator Controller* el cual puede ser modificado desde la ventana de *animator* [20].

- *Audio source:* permite dotar de un archivo de audio al objeto y modificar los distintos parámetros del sonido producido [21].

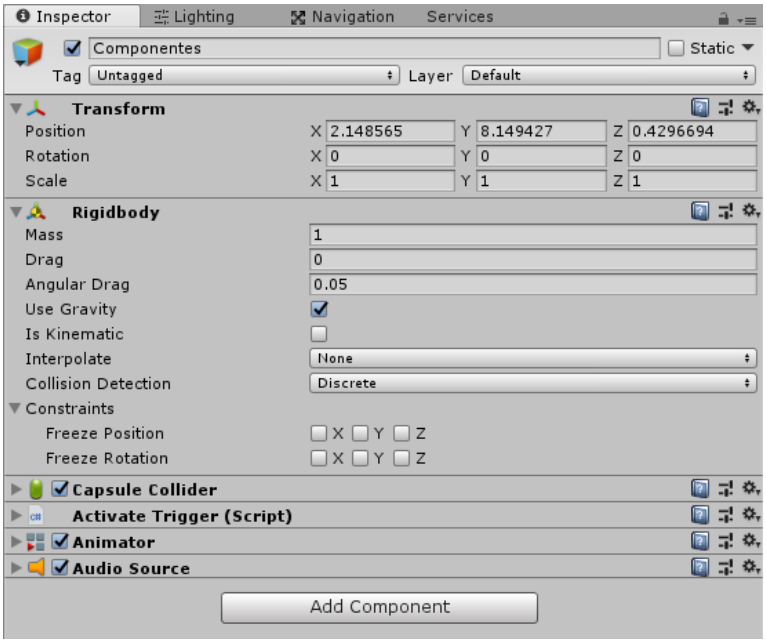

Imagen 6: GameObject con varios componentes

Finalmente, los *GameObjects* pueden ser almacenados en las carpetas del proyecto pasando a llamarse *prefabs*. Los *prefabs* son copias de objetos que se pueden reutilizar, presentando la ventaja de que un cambio realizado en una de las copias puede ser aplicado al resto [22].

4.1.3. **Programación**

Mediante la programación de *scripts* Unity permite extender las funcionalidades de los objetos, como ya se ha mencionado en el punto anterior. El motor es compatible con los lenguajes de programación C#, Boo o JavaScript, siendo el primero de estos el más utilizado respecto a los otros dos que ya están en desuso [23].

Lo que hace que los *scripts* puedan comportarse como un componente más de un objeto es la existencia de la clase *MonoBehavior*, y es que cualquier clase que herede de esta podrá adquirir este comportamiento [24].

Entre las ventajas del uso de los *scripts* destacamos dos. La primera se corresponde con la capacidad de acceder a los diferentes componentes presentes en el *GameObject* al cual pertenece el *script*, utilizando el método *GetComponent<T>*. La segunda es la posibilidad acceder a las variables públicas del propio *script* y modificarlas directamente desde el inspector.

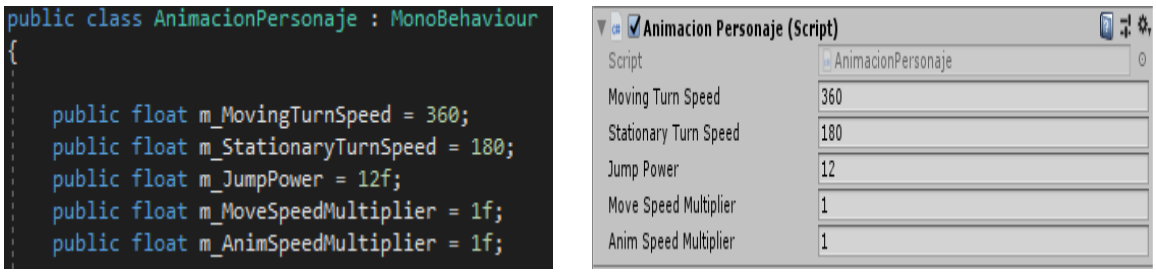

Imagen 7: Variables en script e inspector

4.1.4. **Flujo de ejecución**

Es importante conocer el flujo de ejecución a la hora de desarrollar un proyecto para seguir una organización adecuada y no perjudicar el resultado global. En este apartado se explica el flujo que sigue el motor durante la ejecución, centrándonos principalmente en la clase *MonoBehaviour* que se ha expuesto anteriormente [25].

Durante una ejecución el motor realiza una serie de interacciones con los diferentes objetos activos de la escena y sus componentes. Un *GameObject* desactivado no aparecerá en la escena y por lo tanto no se podrán apreciar los cambios en sus componentes. Sin embargo, pese a encontrarse desactivado es posible acceder y realizar modificaciones que tendrán efecto una vez ese objeto sea activado.

En la Imagen 8 se aprecian las funciones principales y más utilizadas durante la ejecución. Estos eventos pueden ser utilizados para que el objeto realice las acciones deseadas en un punto determinado de la ejecución. Cabe decir que estas funciones no son obligatorias y que no es necesario que todo *script* las posea, por ejemplo podríamos tener un archivo que sirviera para guardar el valor de determinados parámetros y que no utilizará ninguna de estas llamadas.

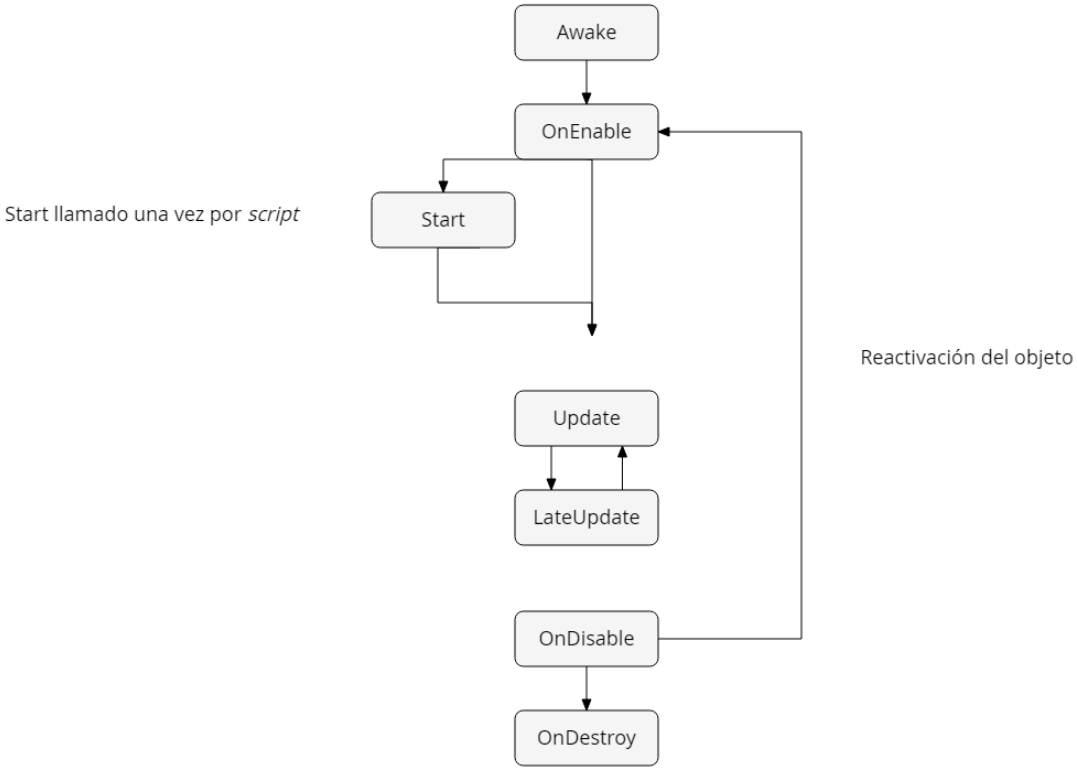

Imagen 8: Diagrama flujo básico MonoBehaviour

A continuación se procede a la explicación de las llamadas más utilizadas de la clase *MonoBehaviour*. Empezamos por las primeras que son ejecutadas en determinados puntos del flujo:

- *Awake*: esta función es llamada antes de cualquier *Start* y antes de la instanciación de un prefab (si el *GameObject* está inactivo no se realiza la llamada hasta que se activa).

- *OnEnable:* función llamada tras la activación del objeto. Esto sucede cuando una instancia *MonoBehaviour* es creada, en la carga de un nivel o cuando un *GameObject* con un *script* es instanciado.

- *Start:* esta función es llamada antes del primer *frame* de la función *update* solo si esta activa la instancia al *script*.

- *OnDisable:* llamada cuando el objeto es desactivado o antes de que se produzca la llamada a la función *OnDestroy*.

- *OnDestroy:* función llamada después de todos los *frames* de *update* y durante el último *frame* de existencia del objeto. El objeto será destruido ante la llamada *Object.Destroy* o al finalizar la escena.

Las siguientes funciones son ejecutadas durante el periodo de vida del objeto y permiten al programador modificar el comportamiento y los componentes del objeto en tiempo real. Las más utilizadas son las siguientes:

- *Update:* se trata de la función principal de actualización y es llamada una vez por *frame.* Su utilidad es muy grande y da mucho juego al programador, pero se debe de tener cuidado ya que un mal uso podría generar una sobrecarga y tener efectos negativos.

- *LateUpdate:* función llamada una vez por fotograma tras la finalización de *Update*, esto quiere decir que todos los cálculos que se especificaron en *Update* y habrán sido realizados. Un ejemplo de uso podría ser el cálculo de la posición del personaje en *Update* y realizar el movimiento de la cámara en *LateUpdate*.

Las funciones anteriores son las básicas ejecutadas por un *MonoBehaviour*, pero además vamos a mencionar a continuación las que intervienen en el funcionamiento de los *colliders* dado que son un parte esencial de las interacciones. Entre ellas encontramos los siguientes grupos:

- *OnTriggerEnter, OnTriggerExit, OnTriggerStay:* funciones activadas cuando interaccionan dos *colliders* y uno está marcado como *trigger*, lo cual provoca que este pueda ser atravesado. Estas funciones detectan la entrada, la salida y la estancia en el *collider*, siguiendo el orden en el que se listan.

- *OnCollisionEnter, OnCollisionExit, OnCollisionStay:* funciones similares a las de tipo *trigger* con la diferencia de que los *colliders* producen colisión cuando se encuentran y no se atraviesan.

Se han expuesto las funciones y las llamadas principales del flujo, con las que es posible ser consciente del funcionamiento y trabajar en consecuencia a esto. Por otro lado, existen más funciones que intervienen en aspectos del flujo como pueden ser el renderizado o las corrutinas.

4.2. **Editor**

En el editor de Unity es donde tiene lugar la mayor parte del desarrollo. A través de él se elaboran las escenas y se administran los diferentes componentes. Además, permite la ejecución del proyecto de forma sencilla sin necesidad de exportar el videojuego. Por último, todas las ventanas del editor pueden personalizarse y adaptarse al gusto del usuario, presentando así la capacidad de ajustarse a los distintos tipos de trabajo.

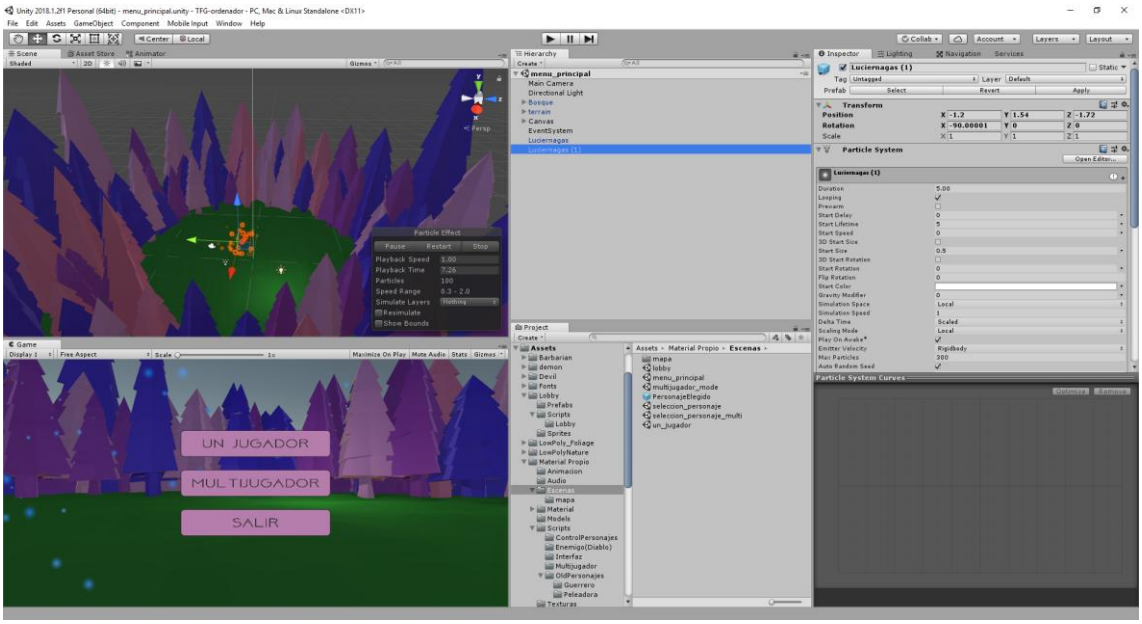

Imagen 9: Ventanas principales del editor

Los siguientes puntos introducen las diferentes ventanas de Unity, empezando por las básicas para posteriormente introducir las que contienen una funcionalidad más avanzada.

4.2.1. **Ventanas principales**

Cuando ejecutamos nuestro primer proyecto en Unity se nos presentan una serie de ventanas básicas listas para utilizar. En este punto, se introducen las ventanas principales y la utilidad de cada una.

Escena

Ventana que muestra la escena de trabajo actual en la cual podemos interactuar con los diferentes elementos de la misma. A través desde este panel podemos realizar directamente operaciones de escalado, traslado o rotación sobre los objetos, obteniendo resultado visual inmediato tras los cambios efectuados.

Juego

Se trata del renderizado de las cámaras del juego, mostrándolo en su estado actual y permitiendo la interacción con él desde esta ventana.

Jerarquía

Contiene cada elemento de la escena y nos permite añadir, eliminar, copiar y duplicar los objetos. Este panel tiene la peculiaridad de permitir relaciones de parentesco en las que un objeto puede tener hijos, de forma que estos queden ligados al padre [26].

Inspector

Esta venta muestra los componentes y propiedades del objeto seleccionado, permitiendo realizar modificaciones sobre estos [27].

Proyecto

Ventana que permite acceder a las diferentes carpetas del proyecto y gestionar los *assests*, que es la forma con la que Unity llama a todos los materiales utilizados. Esta ventana contiene las siguientes funcionalidades:

- Nos permite crear copias de *prefabs* en la escena simplemente arrastrando el elemento deseado al panel de jerarquía o a la escena.
- Permite añadir componentes a un *gameobject* arrastrando el *asset* al inspector del objeto.
- Al arrastrar un elemento que no se encuentre en la estructura a la ventana de proyecto se crea un nuevo *prefab* con el cual podemos realizar las funciones anteriores.

Consola

Permite visualizar los mensajes lanzados por el compilador y los scripts.

4.2.2. **Ventanas avanzadas**

Además de contar con las ventanas básicas que se han descrito en el punto anterior, Unity nos proporciona una serie de funcionalidad más avanzada y que no encontramos a simple vista. Esta serie de herramientas nos permiten llevar el desarrollo de nuestro producto a otro nivel y podemos acceder a ellas desde los distintos menús.

Iluminación

Se trata del punto principal de control de la iluminación global. Pese a que el motor ofrece unos resultados buenos con la configuración por defecto, desde este panel se permite el ajuste de diversos parámetros, ofreciendo de este modo la posibilidad de optimizar aspectos como la calidad, la velocidad o el almacenamiento de nuestro videojuego [28].

Navegación

El motor contiene un sistema de navegación que permite la creación de personajes que se muevan inteligentemente por el mapeado gracias al uso de mallas de navegación. A través de esta ventana se pueden controlar los parámetros que intervienen en este proceso.

Servicios

Ventana que contiene una serie de servicios integrados que proporciona Unity al usuario, como pueden ser el servicio multijugador o el de anuncios.

Animator

En este caso nos encontramos ante la ventana que, mediante el uso de una máquina de estados, nos permite realizar la animación de un objeto. Gracias a los parámetros de distintos tipos que se nos proporcionan, es posible realizar las transiciones entre las animaciones de varias formas ofreciendo flexibilidad a la hora de trabajar [29].

Profiler

Se trata de una ventana que nos muestra los recursos utilizados en las distintas áreas del juego, permitiéndonos de esta forma realizar una optimización basándonos en datos de uso [30].

Asset store

Sección que nos permite acceder a la tienda de Unity desde el propio editor. Mediante su uso se permite la interacción con la tienda dando la opción de descargar e importar paquetes directamente a nuestro proyecto [31].

5. Diseño del videojuego

En esta sección se unen las diferentes partes que han dado como resultado el videojuego, de este modo se conocerán los detalles del juego desde los primeros conceptos hasta el resultado final.

5.1. Concepto

La idea principal del proyecto ha sido desarrollar un videojuego del género horda en el cual hay que superar un número limitado de oleadas de enemigos. El progreso durante las rondas va seguido de un aumento de la dificultad del juego que modifica los enemigos y sus características, creando de esta forma la sensación de asedio propia de un juego de este estilo.

El propósito principal sitúa a dos jugadores en la acción los cuales deben decir si cooperar o eliminar al otro participante. Esta decisión afecta en la cantidad de puntuación que obtienen los jugadores, ofreciendo una delgada diferencia y premiando más la eliminación del otro jugador. De esta forma se consigue añadir otro factor de tensión que lleve al jugador a tomar decisiones en función de su personalidad, creando un dilema añadido a la presión de los enemigos.

El juego tiene lugar en una pequeña área limitada que facilita el encuentro entre ambos jugadores y los enemigos, los cuales pueden aparecer en cualquier sitio del mapa y sorprender al jugador.

Como se deduce de lo anterior el juego busca innovar en el género añadiendo el factor decisión sobre la cooperación, obligando al jugador a que tenga un debate interior acerca de si es correcto acabar con aquel que te ha ayudado durante la trayectoria de la partida para así obtener mayor recompensa.

En el aspecto comercial la plataforma objetivo es Android, puesto que existen muy pocos productos del estilo en ella. Al orientarse a esta plataforma móvil se busca que las partidas sean rápidas y que el control sea sencillo a la vez que intuitivo.

5.2. Ambientación

El siguiente punto introduce la historia sobre la que está basado el videojuego y tras su lectura los diferentes diseños que aparecen a lo largo del juego cobran sentido. A continuación, se introduce la historia que da vida al juego.

En el reino de Ryhm un caballero real se encuentra en una persecución tras una campesina que se ha dado a la fuga después de robar uno de los libros prohibidos de palacio. En una casa a las afueras del reino, el caballero logra atraparla tras abalanzarse sobre ella, provocando la caída del libro prohibido. Una vez inmovilizada la mujer, ambos reparan en el libro, el cual se encuentra abierto en el suelo, mostrando un pergamino que no pueden evitar leer. Al poco tiempo ambos caen desmayados.

Algo no va bien, nuestros dos protagonistas despiertan en una explanada extraña, rodeados de arboles de colores que nunca antes habían visto, de pronto, recuerdan el libro y la información que contenía acerca de un bosque mágico.

El pergamino encontrado narraba la existencia de un bosque mágico situado cerca de una de las puertas al infierno, según habían leído este hecho era el culpable de todos los sucesos fuera de lo común que se producían en el bosque. Más allá de verse atemorizados por el propio bosque, eran sus habitantes los que les preocupan.

Fruto de la magia que brotaba de la puerta al infierno, una colonia de arañas reales fue dotada de grandes poderes y fuerza, hasta tal punto que habían sometido a los diablos que rondaban la puerta, utilizándolos de esclavos.

Aquel ruido que parecía lejano cada vez se siente más cerca, ambos saben que alguien no quería que se revelara esa información y ahora van a experimentar en sus propios cuerpos lo que tras leer el pergamino parecía imposible.

5.3. **Diseños**

El presente punto contiene todos los diseños y modelos utilizados en el desarrollo del videojuego. Seguidamente se exponen las distintas partes del videojuego y las peculiaridades que presentan.

Mapa

La acción tiene lugar en un terreno cerrado del cual los jugadores no pueden salir. El diseño, Imagen 10, está inspirado en un bosque mágico y la intención es recrearlo jugando con árboles de diferentes tamaños y colores, iluminación que nos ponga en situación y partículas que nos transmitan la sensación de encontrarnos en el lugar de la historia.

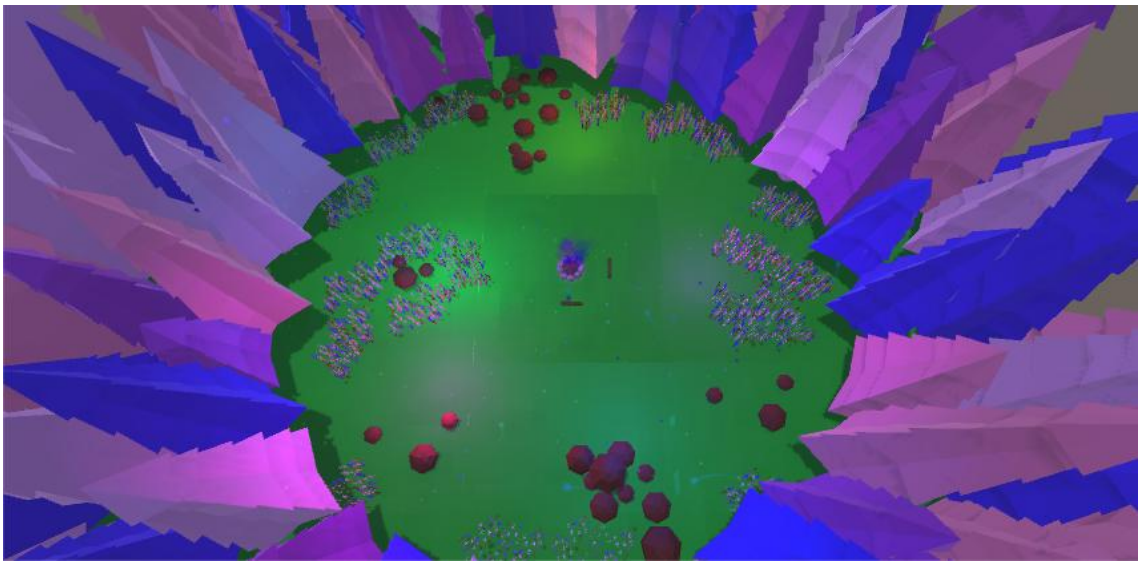

Imagen 10: Mapa de juego

Personajes

Respecto a los personajes, el jugador tiene la posibilidad de elegir entre dos opciones diferentes. Ambos personajes son los protagonistas de la historia y por lo tanto se ofrece la opción de jugar como el caballero o como la campesina. Esta elección afecta en la partida puesto que dispondremos de diferentes habilidades en función del campeón elegido. A continuación, se muestran ambos personajes explicando cada una de sus características.

Campesina

La campesina, Imagen 11, presenta un ataque principal basado en un combo de golpes cuya animación puede finalizar tras cada golpe si no se vuelve a presionar el botón correspondiente. Los tres ataques se producen rápidamente y siguen el orden de puñetazo derecho, puñetazo izquierdo y patada.

La segunda habilidad de la campesina le otorga la capacidad de recuperar un porcentaje de su vida. Tras activarla se puede visualizar el efecto pertinente en la barra de salud y se puede presenciar una serie de partículas verdes alrededor del personaje que nos indican que hemos utilizado la habilidad. Esta habilidad tiene un tiempo de enfriamiento de 5 segundos.

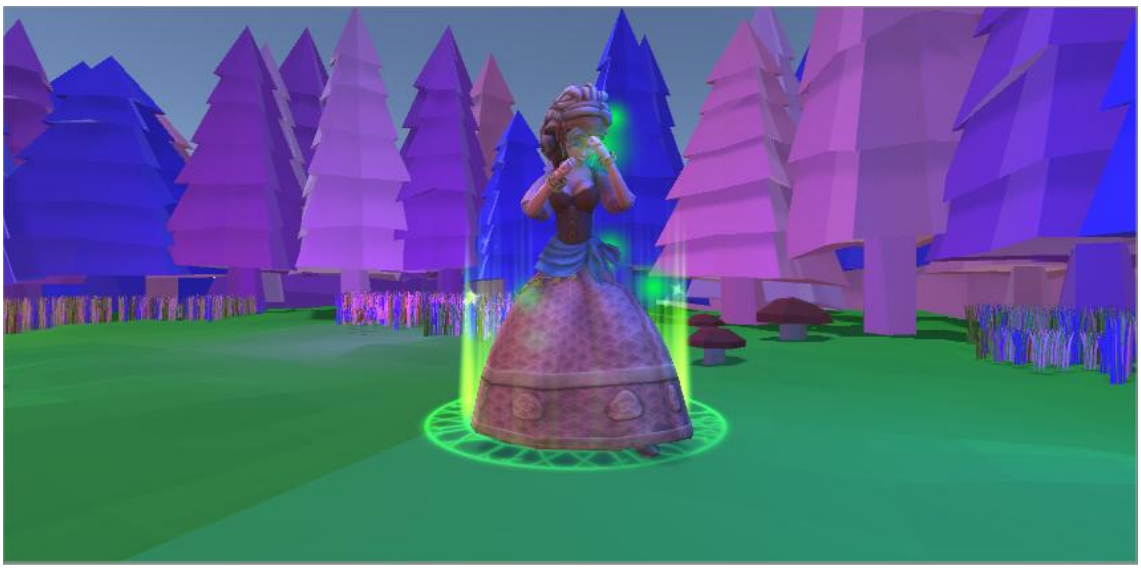

Imagen 11: Modelo de campesina con efecto de habilidad

Caballero

Al igual que la campesina, el ataque principal del caballero se trata de uno de tipo cuerpo a cuerpo que consiste en una sucesión de dos golpes con la espada. El primer golpe del combo se trata de un ataque frontal con la espada, mientras en el segundo el caballero da una vuelta y realiza su ataque. La animación del primero se puede cortar y realizar el segundo rápidamente para ganar velocidad.

La habilidad especial del caballero, Imagen 12, le otorga una subida de daño temporal durante 5 segundos. Al activar la habilidad un torbellino de partículas rojas rodearan al personaje, indicándonos que el efecto de la habilidad está presente. Cuando finaliza el tiempo de la mejora las partículas desaparecen y la habilidad no se puede volver a utilizar durante 7 segundos.

Imagen 12: Modelo de caballero con efecto de habilidad

Las estadísticas de ambos personajes son iguales, encontrando diferencia solamente en sus habilidades y ataques. La vida que tienen es de 100 unidades, en el caso de la campesina esta no podrá sobrepasar este valor con su cura. Por otra parte, el ataque de ambos es de 50 unidades, el caballero aumenta este valor en 25 unidades con su habilidad y cuando finaliza el efecto vuelve al valor base establecido.

Enemigos

Durante el desarrollo de una partida pueden aparecer en el mapa dos tipos de enemigos, Imagen 13, y estos se corresponden con la historia. El primer enemigo que nos encontraremos son diablos, los cuales según la historia son esclavizados por el segundo enemigo, las arañas reales.

Los diablos son los enemigos que más aparecen dado que su dificulta es menor. Se caracterizan por ser pequeños y desplazarse volando. Su vida es menor que la de las arañas y la velocidad a la que se desplazan es mayor, esta última característica les añade un punto de dificultad.

Por otra parte, las arañas no las encontraremos hasta que avance la partida ya que se tratan de enemigos más peligrosos para el jugador. Se caracterizan por su gran tamaño y su lentitud. Entre sus características resaltan su gran cantidad de vida y su gran daño, permitiéndole aguantar varios golpes y matar a un jugador de dos ataques.

Las características de ambos enemigos pueden ser manipuladas, cosa que se ve reflejada a medida que avanza la partida y va aumentando la dificultad, dado que los enemigos son más veloces, tienen más vida o presentan un más potente.

Imagen 13: Modelos de araña y diablo

5.4. **Rondas**

Una partida consta de 5 oleadas diferentes cuya dificultad incrementa a medida que progresamos entre ellas. Entre las rondas el jugador encontrará variedad en número y tipo de enemigos, los cuales van mejorando sus propiedades con el avance de la partida. A continuación, se detallan las oleadas listando los enemigos que aparecen en estas así como sus características.

Ronda 1

Tabla 2: Enemigos ronda 1

Enemigo	Características					
	Cantidad	Apariciones	Vida	Daño	Tiempo entre ataques	Velocidad de movimiento
Diablo	10	Cada 4 s.	50 u.	10 u.	2 s.	2 u.

Ronda2

Tabla 3: Enemigos ronda 2

Enemigo	Características					
	Cantidad	Apariciones	Vida	Daño	Tiempo entre ataques	Velocidad de movimiento
Diablo	15	Cada 3 s.	100 u.	15 u.	2 s.	2.5 u.

Ronda 3

Tabla 4: Enemigos ronda 3

Enemigo	Características					
	Cantidad	Apariciones	Vida	Daño	Tiempo entre ataques	Velocidad de movimiento
Diablo	10	Cada 2 s.	100 u.	15 u.	1.5 s.	2.5 u.
Araña	2	Cada 10 s.	200 u.	30 u.	3 s.	0.5 u.

Ronda 4

Tabla 5: Enemigos ronda 4

Enemigo	Características					
	Cantidad	Apariciones	Vida	Daño	Tiempo entre ataques	Velocidad de movimiento
Diablo	10	Cada 2 s.	100 u.	15 u.	1.5 s.	2.5 u.
Araña	2	Cada 10 s.	200 u.	30 u.	3 s.	0.8 u.
Diablo	5	Cada 2 s.	100 u.	25 u.	1.5 s.	2.5 u.

Ronda 5

Tabla 6: Enemigos ronda 5

Enemigo	Características					
	Cantidad	Apariciones	Vida	Daño	Tiempo entre ataques	Velocidad de movimiento
Diablo	15	Cada 2 s.	100 u.	25 u.	1.5 s.	2.5 u.
Araña	1	-	200 u.	50 u.	3 s.	0.8 u.
Diablo	15	Cada 2 s.	100 u.	25 u.	1.2 s.	2.5 u.
Araña	1	-	250 u.	50 u.	3 s.	1 u.
Araña	1	-	300 u.	50 u.	2 s.	1 u.

5.5. **Interfaz**

La plataforma objetivo de nuestro proyecto es Android y es por esto que el diseño de la interfaz se adapta a estos dispositivos, facilitando la interacción al usuario y buscando que su experiencia sea la mejor posible.

A continuación, se muestra la interfaz del videojuego Imagen 14, exponiendo sus diferentes partes así como la función que tienen dentro del juego.

Imagen 14: Interfaz de juego y sus componentes

Barra de vida (1)

Está formado por el componente de interfaz *scrollbar* y muestra la vida del jugador en todo momento. Su valor se actualiza cuando el personaje recibe daño, reduciendo el tamaño de la vida en función del golpe recibido. En el caso de la campesina, cuando realiza su habilidad de curación se produce un aumento proporcional.

Indicador de ronda (2)

Formado por un componente de texto, muestra la ronda en la que va a entrar el jugador, es decir, solo se muestra al inicio de ronda y cuando aparece el primer enemigo desaparece para dejar mayor visibilidad al usuario.

Indicador de puntuación (3)

Formado por un componente de texto, muestra los puntos que se han conseguido en la ronda. Cada enemigo eliminado suma una cantidad al total que se va acumulando.

Joystick (4)

Formado por un *joystick* permite controlar los movimientos del personaje. Con la modificación de su posición se generan unos valores y a partir de estos se modifica la posición del personaje sobre el mapa.

Botones (5)

Están formados por el componente botón de interfaz que nos proporciona Unity. Son los botones que controlan las acciones de los personajes y en concreto tienen las siguientes funciones:

- **Botón A:** se encarga de los ataques del campeón, al activarlo varias veces podremos realizar los combos del personaje jugado.

- **Botón B:** activa la habilidad del personaje que estemos jugando

5.6. Menús

En este apartado se expone la interfaz que se ha diseñado para los menús además de indicar como se conectan entre ellos.

Una interfaz de menú está formada principalmente por una serie de botones que nos permiten navegar entre ellos de forma intuitiva, es decir, nos permiten pasar de una escena a otra en la cual está el siguiente menú. La Imagen 15 muestra un ejemplo del diseño que se ha seguido para los menús. Como podemos observar la disposición de los menús se sitúa sobre el terreno de juego.

Imagen 15: Interfaz de un menú

Una vez mostrado el diseño la Imagen 16 muestra como están conectados los diferentes menús. Desde el menú principal se nos da la opción de elegir la modalidad de juego entre un jugador y multijugador. Independientemente de la opción seleccionada el usuario es dirigido a la ventana de selección de personaje. En el caso de haber elegido el modo solitario, la partida comienza tras elegir nuestro campeón. Por otro parte, en el caso del multijugador aparecerá la ventana de sala donde el jugador podrá crear una partida o unirse a una ya creada.

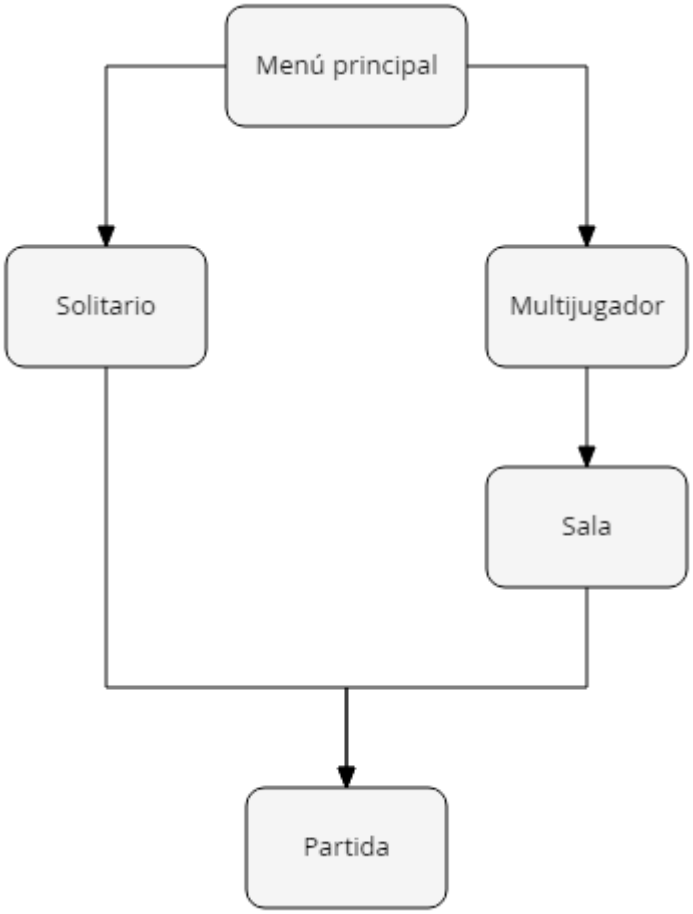

Imagen 16: Diagrama de navegación entre menús

A diferencia del resto de los menús, el de sala (Imagen 17), tiene un funcionamiento un poco más avanzado con el cual un consumidor no habitual de videojuegos puede tener algún problema, es por eso que se explica a continuación.

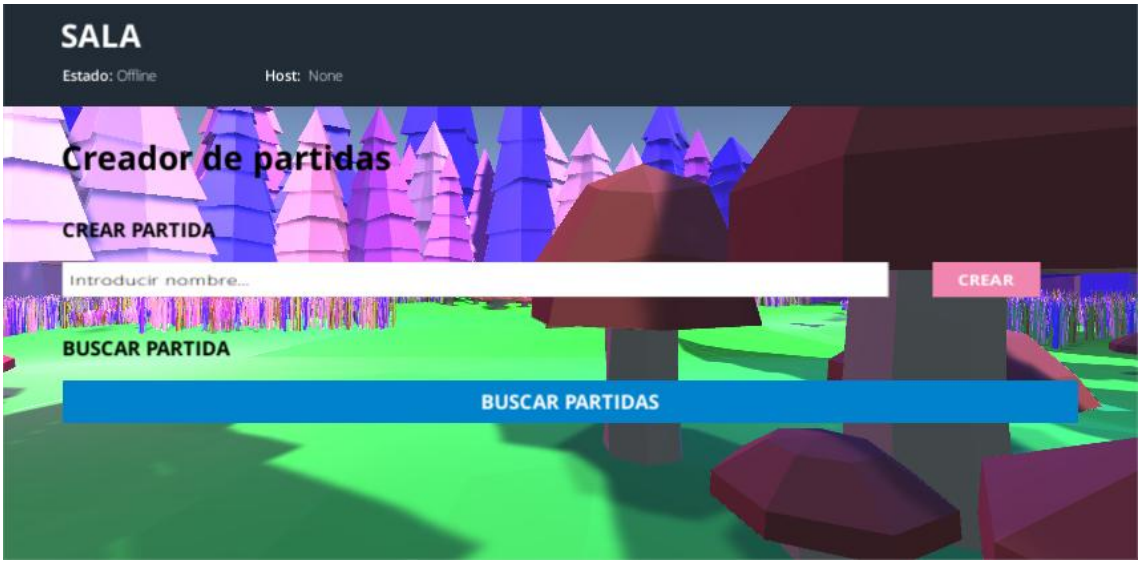

Imagen 17: Captura de la ventana de sala

Cuando el usuario entra en este menú puede ver a simple vista que se le ofrecen dos posibilidades. La primera es crear una partida, para realizar esta acción lo primero es introducir el nombre por el cual se quiere identificar la partida. Una vez realizado esto se pulsa el botón crear y se modifica la pantalla para indicar que se ha creado correctamente y hay que esperar a otro jugador. Por otro lado, la opción de buscar una partida nos lista todas las posibles partidas que se encuentran disponibles para unirse. Tras la unión de dos jugadores a una partida ambos deberán pulsar el botón que indica que están listos para jugar. Una vez los dos jugadores confirman que están listos comienza la partida.

6. Implementación del videojuego

En este punto se expondrán los aspectos más importantes de cómo se han realizado las distintas partes del videojuego, desde los personajes y los distintos componentes que los forman hasta los diversos objetos que controlan aspectos del juego.

La primera idea del proyecto era orientarse únicamente en el apartado multijugador del juego, pero al decidir realizar un modo de juego solitario, hemos optado por dividir el siguiente apartado de forma que se exponga en primer lugar la implementación seguida para el modo solitario y, partiendo de ese punto realicemos una transición hacia el apartado multijugador. De esta forma quedan más claros los conceptos y permite plasmar de una forma bastante clara el paso de un juego de un solo jugador a uno multijugador.

6.1. Modo solitario

Este modo se caracteriza porque el jugador puede disfrutar de una partida en solitario, sin necesidad de conexión a la red. A continuación, se exponen las distintas partes que unidas tienen como resultado el correspondiente modo de juego, listando los diversos aspectos de cada una y profundizando en sus características.

6.1.1. Personajes

Los personajes son los protagonistas principales del proyecto, ya que con ellos interactuará el usuario constantemente. Lo primero que haremos será explicar los componentes básicos que tienen y posteriormente avanzaremos hacia características más avanzadas de ellos como pueden ser los distintos *scripts* que los forman o sus animaciones.

Componentes

Nuestros personajes tienen varios componentes, como podemos ver en la Imagen 18, y en este apartado se explican brevemente aquellos que son esenciales, ya que estos han sido nombrados durante el apartado del motor. Al introducir primero los básicos se obtiene una base para la explicación de los otros componentes más avanzados a los cuales dedicamos apartados separados. A continuación, se exponen los componentes mencionados:

- *Transform*: dota a nuestros personajes de posición, rotación y escala en el mapa. Este componente será modificado por elementos que se explicarán más adelante en este punto.

- *Rigibody:* proporciona al personaje una serie de cualidades que le permiten tener interacciones físicas.

- *Capsule collider:* detector de colisiones de nuestro personaje. Gracias a él los campeones no atraviesan el mapa y pueden chocar con los distintos elementos del mapa que presenten un componente de este tipo.

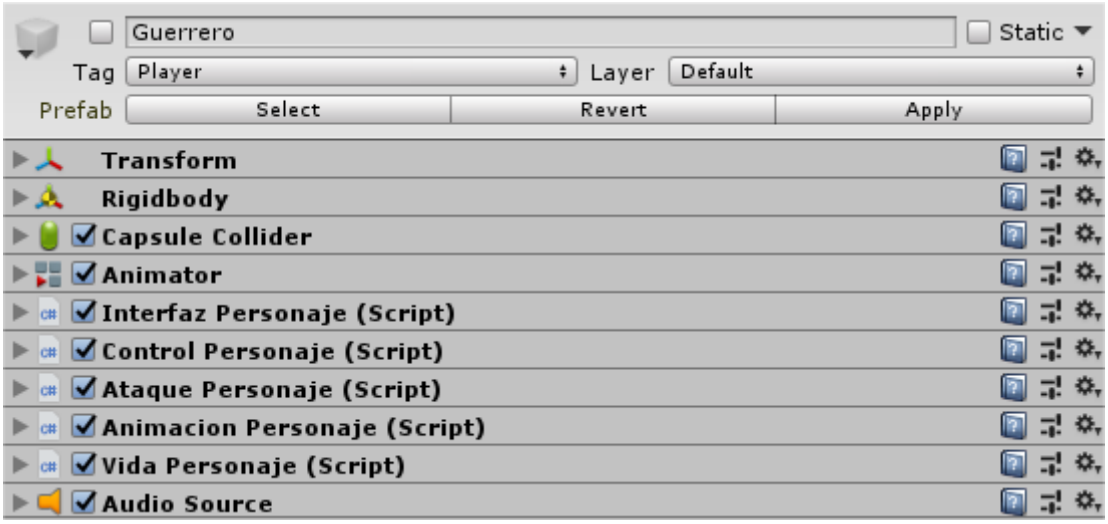

Imagen 18: Componentes de un personaje en el modo solitario

Animación

Nuestros personajes realizan una serie de animaciones gracias a que presentan un componente de animación, este componente se trata del *animator*, el cual lo podemos localizar en la Imagen 18. El *animator* recibe como parámetro un controlador de animaciones, este controlador es el que contiene toda la disposición de animaciones que realiza el personaje y se configura mediante la ventana de *animator*, Imagen 19, cuya funcionalidad está basada en una máquina de estados.

Las animaciones de los personajes se distribuyen en distintos estados, estos estados se encuentran relacionados por transiciones. Una transición puede contener uno o varios parámetros que indiquen el paso de un estado a otro, estos parámetros pueden ser de tipo entero, real, *bool* o *trigger*.

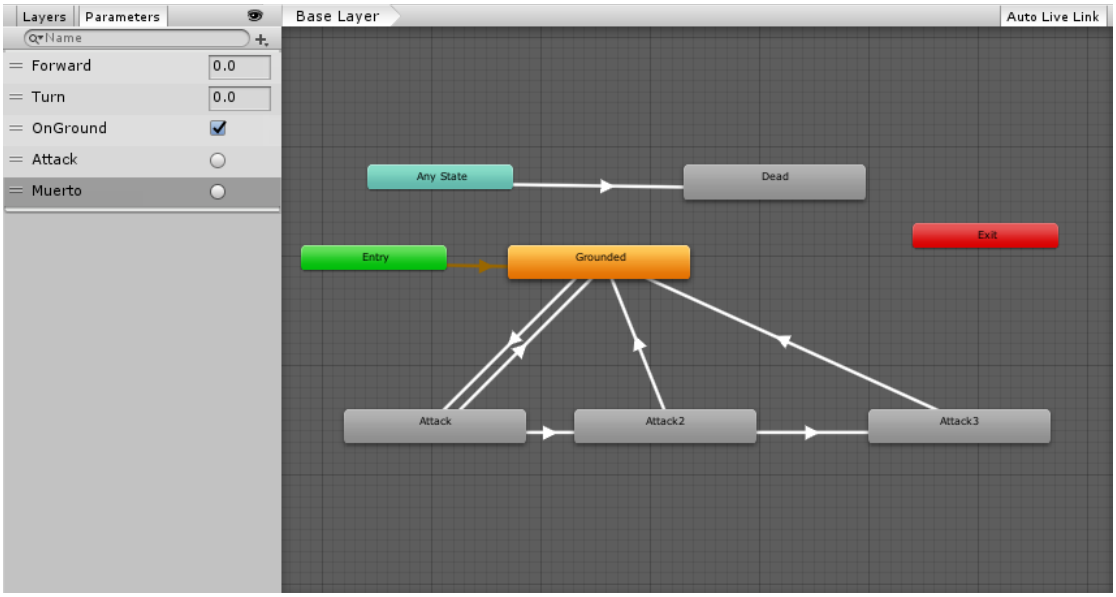

Imagen 19: Máquina de estados con el controlador de la campesina

Los controladores de la campesina y el caballero son ligeramente diferentes, ya que la campesina puede realizar un ataque más durante su combo. A pesar de esto, la explicación que a continuación se realiza se aplica en ambos casos.

Los estados de la máquina que podemos observar en los controladores de nuestros personajes contienen las animaciones y son los siguientes:

- *Entry:* estado inicial de la máquina ejecutado al inicio de esta. Su transición es de color amarillo y marca el estado por el que empieza la máquina.

- *Any state:* estado especial que siempre está presente. Se utiliza para realizar una transición independientemente del estado en el que se encuentre la máquina. En nuestro caso nos sirve para controlar la animación de muerte, la cual se producirá siempre que el personaje se quede sin vida.

- *Grounded:* simboliza que nuestro personaje se encuentra en el terreno y es el encargado de todas las animaciones de movimiento incluyendo giros, cambios de sentido y desplazamientos, como andar o correr.

- *Attack:* los estados que presentan estos caracteres se corresponden con las animaciones de ataque del personaje. En el caso del caballero encontramos dos estados de este tipo y en la campesina tres.

- *Dead:* estado encargado de realizar la animación de muerte del personaje. Como ya se ha mencionado, la máquina llega a este estado cuando el personaje no tiene vida, independientemente del estado en el que se encontrara.

- *Exit:* indica a la máquina de estados que debe finalizar su ejecución. En nuestro caso no es utilizado dado que finalizamos las animaciones cuando muere el personaje realizando una destrucción del objeto.

Por otro lado, como hemos mencionado anteriormente, los controladores presentan una serie de parámetros que controlan las transiciones entre las animaciones. En nuestro caso, los parámetros utilizados son los siguientes:

- *Forward:* parámetro de tipo *float* que indica la velocidad de movimiento de nuestro personaje, alternando entre las distintas animaciones como correr o andar.

- *Turn:* parámetro de tipo *float* que controla la velocidad de giro de nuestro personaje.

- *OnGround:* parámetro de tipo *bool* que indica si nuestro personaje se encuentra en tierra, en caso de haber implementado un salto también habría servido para sus animaciones.

- *Attack:* parámetro de tipo *trigger* que cuando es activado realiza una transición al primer estado de tipo *attack*. Sus sucesivas activaciones hacen que recorra los distintos estados haciendo posible la visualización de un combo de golpes. En caso de no volverse a activar se vuelve al estado *grounded*.

- Muerto: parámetro de tipo *trigger* que se activa cuando el personaje no tiene vida. Su activación implica ir al estado *dead* independientemente del estado en el que se encuentre la máquina.

Los distintos parámetros que se han expuesto son manipulados mediante código, de forma que se realicen las transiciones entre estados en el momento deseado. La unión de los distintos estados y la existencia de los parámetros diseñados crean el sistema de animaciones de nuestros personajes.

Scripts

Todas las acciones que pueden realizar los personajes están controladas mediante ficheros de código escritos en C#. Gracias a la clase *MonoBehaviour* estos *scripts* son aceptados por los *GameObjects* de los personajes y tratados como un componente más.

Como podemos observar en la Imagen 18, los personajes presentan muchos componentes del tipo *script*. Estos componentes controlan los distintos aspectos de un personaje dotándolos de propiedades que no pueden adquirir con otro componente distinto. Seguidamente se exponen los distintos *scripts* explicando la funcionalidad que proporcionan.

AnimacionPersonaje

En este fichero podemos encontrar los procesos que actúan sobre el animador y sus distintos parámetros gracias a una serie de funciones. La primera se encarga de transformar unos datos globales de movimiento en unos locales, calculando de esta forma cuanto se debe avanzar o girar en la dirección deseada. La segunda función recibe los resultados de la primera y su misión es actualizar las animaciones. Para realizar esta tarea, se modifican los parámetros de movimiento, anteriormente mencionados al tratar las animaciones, de forma que las animaciones se adapten a los cálculos realizados. Finalmente, encontramos la función que cambia el parámetro *attack* del animador, de forma que se realice la animación de ataque cuando el usuario presiona el botón correspondiente.

ControlPersonaje

Partiendo de la cámara utilizada y la posición del personaje realiza una serie de cálculos para obtener la siguiente posición del personaje. Una vez obtenidos los datos le proporciona estos a AnimacionPersonaje.cs para que mueva al personaje y realice las animaciones pertinentes.

InterfazPersonaje

Encargado de obtener la información que proporciona el *joystick* y proporcionársela al *script* ControlPersonaje.cs para que realice sus operaciones.

VidaPersonaje

Este *script* es el encargado de almacenar el valor de vida del personaje y contiene una serie de funciones que ejercen modificaciones sobre este. Se producen constantes actualizaciones para tener el valor correcto de la vida y que la información mostrada por la interfaz sea coherente, ya que los otros *scripts* pueden acceder a las funciones y provocar cambios. En concreto, para el modo solitario son los enemigos los que modifican la vida del personaje cuando atacan.

En el fichero encontramos dos funciones diferentes. La primera es la que permite que el personaje reciba daño, es decir, proporciona la funcionalidad de restar un valor a la vida actual del personaje jugado. Por último, encontramos la función de muerte la cual es llamada cuando la vida del personaje es igual o menor a cero. Su activación provoca el cambio en el parámetro correspondiente del animador y que otros *scripts* comiencen el proceso de fin de partida.

AtaquePersonaje

La función de este *script* es la de realizar daño a los enemigos. Funciona de manera que, cuando el jugador realiza un ataque se produce una comprobación de los objetos presentes en el rango de nuestro personaje, tarea llevada a cabo por un componente que se explica en el siguiente subapartado. Todo enemigo localizado en rango durante un ataque recibe daño a través de su propio *script* de vida.

Objetos hijos

Nuestros personajes realizan una serie de acciones que requieren de la adición de un objeto hijo, concretamente dos, uno que gestiona el rango de ataque y otro que activa los efectos del campeón. El primero lleva una funcionalidad un poco más avanzada que se explica a continuación. Por otro lado, el *GameObject* que contiene los efectos tiene como finalidad activarse cuando se presione el botón correspondiente a la habilidad especial del personaje jugado.

En la Imagen 20 podemos observar el rango de ataque del caballero. Este objeto está formado por un *sphere collider* de tipo *trigger*, es decir, en vez de realizar colisiones con otros objetos manda señales cuando es atravesado.

El objeto tiene un componente de tipo *script* que utiliza dos de las funciones del *collider*. Su funcionamiento es tal que, mediante la función *OnTriggerEnter()* identifica al enemigo que ha entrado en rango y el componente de este que hace referencia a la vida, además de indicar mediante un *bool* que el enemigo está en rango. Por otro lado, la segunda parte de su funcionalidad utiliza la función *OnTriggerExit()* para dejar los valores manipulados por la función de entrada a nulo.

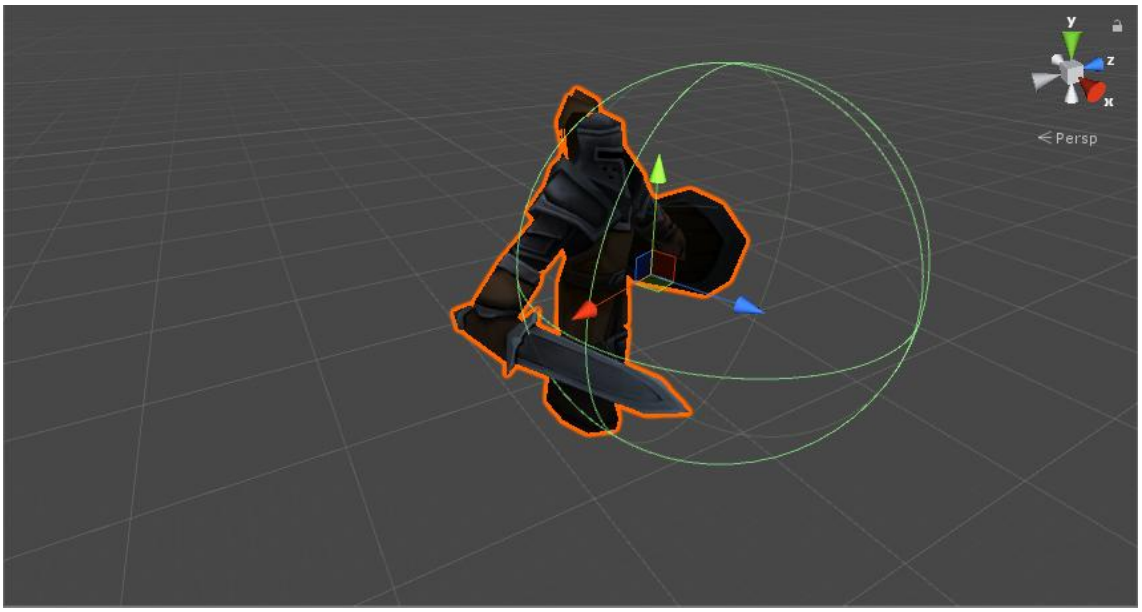

Imagen 20: *Sphere collider* utilizado para calcular el rango de ataque

6.1.2. **Enemigos**

Los enemigos en este videojuego son el principal objetivo de los jugadores y es por ello que adquieren un carácter importante. En el presente apartado veremos las distintas partes que los forman y cómo funcionan.

Componentes

Al igual que los personajes, los enemigos están formados por componentes diferentes, Imagen 21, y en ellos podemos encontrar los básicos que ya se han explicado para el caso del personaje, es decir, un *rigidbody*, un *transform* y un *capsulle collider*. Es por ello que a continuación se exponen aquellos diferentes.

El componente más destacado de los enemigos es el *nav mesh agent* el cual permite que los enemigos persigan a los personajes. Este componente detecta la malla generada del mapa de juego y ofrece una serie de funciones para trabajar sobre esta. En este caso particular, detectamos las posiciones del personaje sobre la malla y vamos guiando al enemigo hacia él, de esta forma conseguimos que el personaje sea seguido satisfactoriamente. Por último, cabe mencionar sobre este componente que nos brinda la posibilidad de modificar la velocidad a la que el sujeto se mueve sobre la malla, gracias a ello en las siguientes explicaciones vemos como este hecho nos permite dar un cambio a la dificultad.

Finalmente, a diferencia de los personajes, los enemigos tienen entre sus componentes otro detector de colisiones, el cual se corresponde con el detector de rango de los personajes. Esta peculiaridad se debe a que la funcionalidad del rango se aplica en todos los *colliders* del objeto, cosa que en el caso de los personajes generaba problemas dado que cabía la posibilidad de que siendo rodeado el personaje, este atacará los enemigos que estaban situados detrás, ya que el *collider* físico recibía interacciones. Por parte del enemigo este problema no sucede, esto es debido a que los enemigos siempre están orientados mirando al personaje y no existe la posibilidad de que reciban un ataque de la espalda del enemigo.

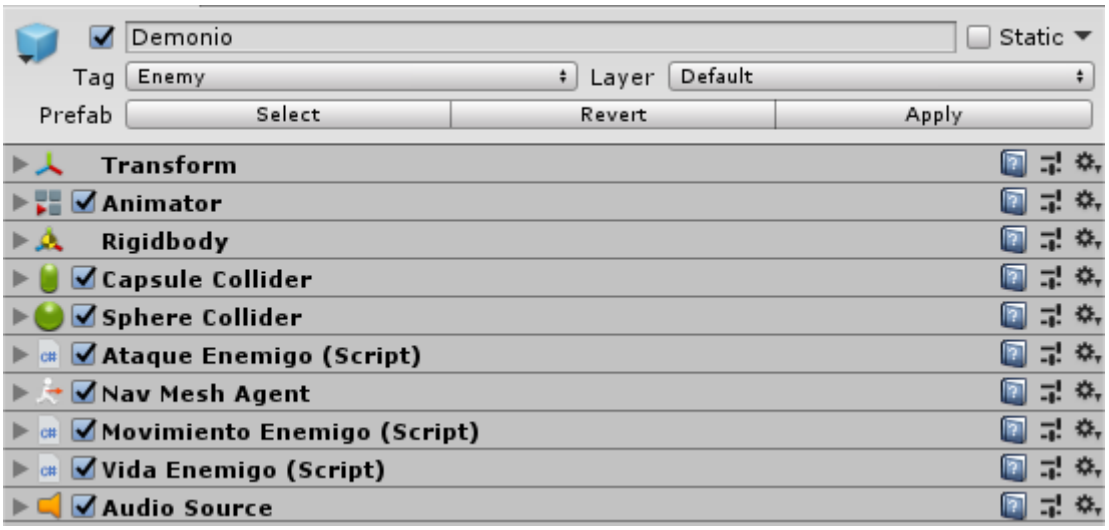

Imagen 21: Componentes de un enemigo en el modo solitario

Animación

Para realizar las animaciones se ha utilizado el componente *animator*, al igual que con los personajes. En el caso de los enemigos, el controlador de animaciones es distinto, como podemos ver en la Imagen 22, y tienen una serie de estados y transiciones diferentes al visto anteriormente, Imagen 19.

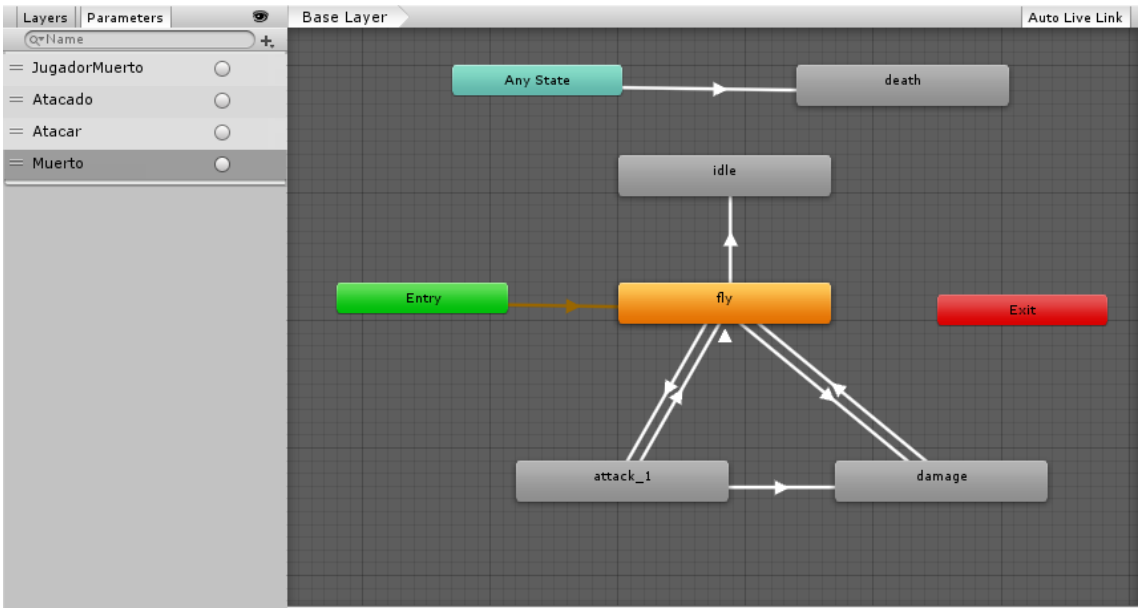

Imagen 22: Máquina de estados con el controlador de los enemigos

Seguidamente se explican los estados de la máquina correspondientes a los enemigos, obviando aquellos estados estándar que ya se han explicado.

- *Idle:* estado que contienen la animación utilizada cuando el enemigo no tiene un objetivo al que seguir, es decir, cuando el personaje que se está jugando pierde toda su vida y muere.

- *Fly:* se trata del estado que realiza la animación de movimiento de los enemigos y al igual que en los personajes es el primero que se llama, como se puede apreciar por la transición entre el estado de entrada y este estado.

- *Attack:* estado contenedor de la animación correspondiente al ataque.

- *Damage:* este estado contiene la animación ejecutada cuando el enemigo recibe daño.

- *Dead:* igual que en el caso del personaje, contiene la animación de muerte y se puede llegar a él desde cualquier estado.

El siguiente paso es explicar los parámetros del animador y los efectos que estos tienen sobre la máquina de estados. A continuación, se exponen los diferentes parámetros que podemos observar a la izquierda de la Imagen 22.

- JugadorMuerto: parámetro de tipo *trigger* que se activa cuando el enemigo no tiene ningún objetivo al que seguir. Permite realizar la transición del estado *fly* al estado *idle*.

- Atacado: parámetro de tipo *trigger* que activa el estado de *damage*. Este parámetro se activa cuando el enemigo recibe daño, es decir, el valor de su vida es reducido por un ataque.

- Atacar: parámetro de tipo *trigger* que se activa cuando el enemigo realiza un ataque, permitiendo de este modo la transición del estado *fly* al estado *attack*.

- Muerto: parámetro de tipo *trigger* activado cuando el enemigo pierde toda su vida, es decir, el valor de esta es menor o igual a cero.

De forma similar a los personajes, la activación de las animaciones de los enemigos será controlada por varios *scripts*, de forma que se consiga "dar vida" correctamente a las criaturas que el jugador enfrentará.

Scripts

Las acciones que realizan los enemigos se ven controladas por tres *scripts* diferentes, Imagen 21. Al igual que los personajes, los ficheros forman parte de los componentes del *GameObject* y les dotan de propiedades únicas que otros componentes no pueden proporcionar. Seguidamente se exponen y explican las peculiaridades de estos *scripts*.

MovimientoEnemigo

Este *script* tiene la finalidad de dotar de movimiento al enemigo. Dado que el jugador es el objetivo, nuestra meta es lograr que el enemigo consiga perseguir al personaje satisfactoriamente. El fichero esta implementado de forma que durante la llamada a la función *Awake()* se localiza al jugador y se obtiene la vida y el objeto de este. Una vez recopilada la información necesaria, mediante la función *Update()* se realizan comprobaciones sobre las vidas del enemigo y del jugador. En caso de que el valor de ambas vidas sea superior a cero, se procede a utilizar la función *SetDestination()* del componente *nav mesh agent* del enemigo, logrando de esta forma que el enemigo se desplace detrás del jugador.

AtaqueEnemigo

El *script* de ataque de un enemigo tiene funcionalidad similar al de un personaje. Como se ha mencionado en los componentes, el objeto del enemigo contiene los dos *colliders* y es por eso que se realiza el cálculo del rango de ataque en este *script*, sin la necesidad de utilizar un objeto hijo como fuera el caso del personaje. Por otra parte, hay presente una función dedicada a detener el ataque del enemigo, en caso de que este sea golpeado, y de este archivo se producen los cambios en el controlador de animaciones correspondientes al ataque, es decir, se activa el *trigger* que dispara la animación correspondiente.

VidaEnemigo

Se trata del *script* encargado de guardar la vida del enemigo y las respectivas funciones que ejercen cambios sobre esta. De manera similar a los personajes, existe una función que permite recibir daño al enemigo. Cuando esta se activa, la vida del enemigo se reduce en función del daño y además se modifica el parámetro de animación *atacado* para que se produzca la animación pertinente. Finalmente, durante la función de muerte, activada cuando la vida se ve reducida a cero, se activa la función desaparecer destruyendo el objeto del enemigo, utilizando

la función *Destroy()* de la clase *GameObject*, y se procede a sumar los puntos que aporta el enemigo al marcador, haciendo uso de una funcionalidad explicada en un punto posterior.

6.1.3. **Cámara**

En este videojuego la cámara es un componente muy importante en la experiencia del jugador. Al tratarse de un producto orientado a la plataforma Android, nos encontramos ante el problema de espacio que supone trabajar en la pantalla de un dispositivo móvil. El reducido espacio hace que sea difícil incluir dos *joysticks*, uno para controlar el personaje y otro para mover la cámara de juego. Además, la inclusión del *joystick* dedicado a la cámara supondría para el jugador la obligación de desatender los botones de habilidad y ataque, cosa que puede llegar a resultar incomodo y empeorar la experiencia del jugador.

La solución encontrada a los problemas anteriormente mencionados ha sido la liberación de la responsabilidad del manejo de la cámara por parte del jugador. El movimiento de la cámara se ve afectado por un componente de tipo *script* añadido a la cámara, el cual se encarga de realizar los movimientos deseados, en concreto, dispone la cámara de forma que siga un estilo de tercera persona fijado a la espalda del jugador, es decir, durante el juego el usuario verá únicamente la espalda del personaje que está utilizando, como se puede ver en el *preview* de la Imagen 23.

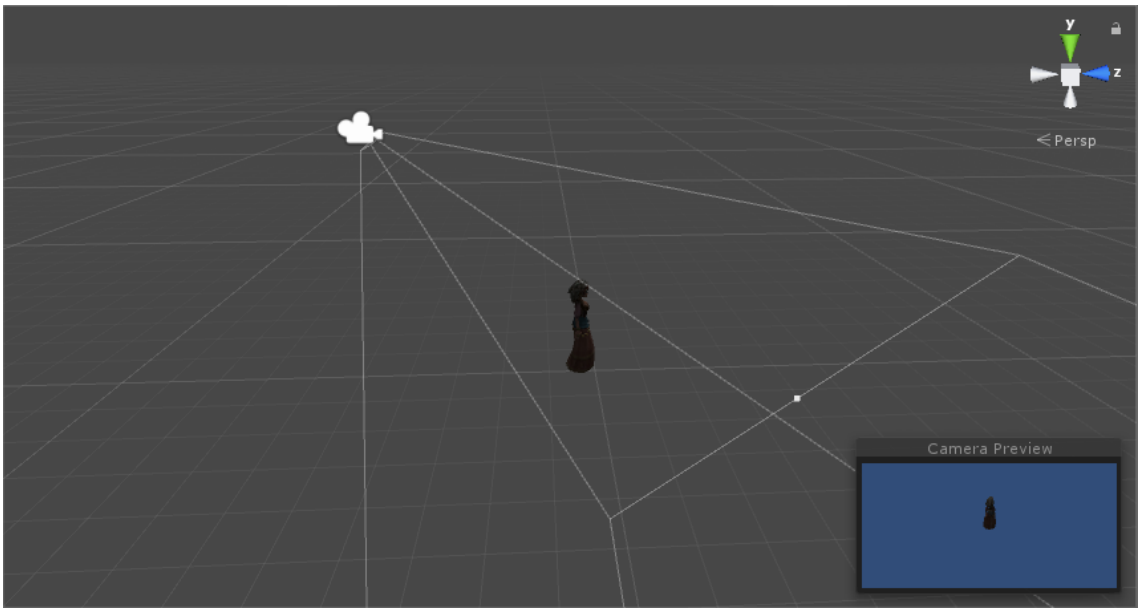

Imagen 23: Cámara en tercera persona del juego

El siguiente paso es comprender la funcionalidad que ofrece el *script* que tienen como componente la cámara, este se expone a continuación.

“Camara”

El presente *script* tiene la función de situar la cámara en un punto y que esta siga a un objetivo. La primera acción que realiza es buscar al personaje durante la función *Start()*, para encontrarlo utiliza el sistema de etiquetas de Unity y busca por la que tiene el título *Player*, una vez localizado el personaje pasa a ser el objetivo. Una vez localizado el objetivo, se producen actualizaciones continuas realizando dos funciones diferentes. La primera función que se realiza es el cálculo de la posición, la cual se obtiene a partir de la posición de la cámara respecto al objetivo. La otra función calcula la rotación de la cámara mediante la técnica *lookAt*, la cual indica a la cámara a qué punto debe orientarse.

Cuando el *script* es añadido como componente es posible modificar una serie de parámetros que faciliten la configuración de la cámara, como se aprecia en la Imagen 24. Entre los ajustes podemos definir la separación de la cámara respecto a su objetivo, en el caso de nuestro juego se utiliza una separación de 5 unidades en el eje Y y -4 unidades en el eje Z. Otras opciones son la interpretación de la separación, que puede ser global o sobre la propia cámara y la opción de calcular la orientación mediante la técnica de *lookAt*.

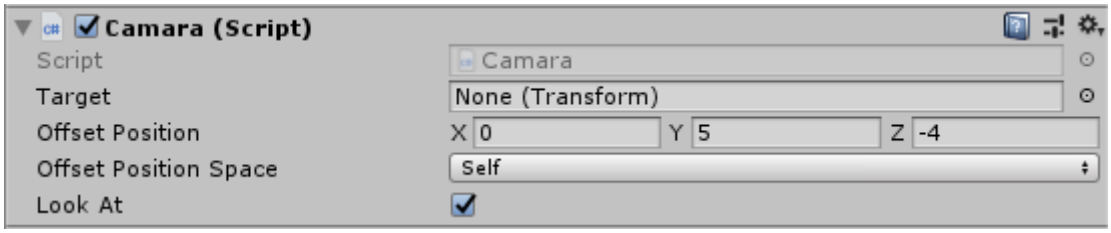

Imagen 24: Valores *script* cámara en el inspector

Gracias a la configuración de la cámara y el *script* pertinente logramos solventar los distintos problemas que se han expuesto durante el punto. El jugador no tiene que centrarse en el manejo de la cámara y puede limitarse al control del resto de los controles, logrando de este modo una mejor experiencia durante el uso del producto.

6.1.4. **Interfaz**

Durante el apartado de diseño ya se ha realizado una exposición de las diferentes partes de la interfaz así como la estética que presenta. En el siguiente punto se introducen los componentes de tipo *script* que hacen posibles las funciones que realizan los componentes de la interfaz. Los ficheros utilizados son presentados seguidamente.

BotonA

Este *script* recoge la pulsación del botón A y es el responsable de desencadenar todas las acciones del ataque de un personaje. Durante la función *Start()* busca al personaje por su etiqueta *Player* y obtiene de este el componente que se corresponde con el *script* de ataque. La pulsación del botón correspondiente en la interfaz se recoge mediante la función *OnPointerClick()*, cuando esta es llamada se utiliza el componente de ataque anteriormente recuperado para ejecutar la función ataque e iniciar todos los procesos que completan el ataque.

BotonB

Script que recoge la interacción del usuario con el botón B y es el encargado del funcionamiento de la habilidad especial de cada personaje. Al igual que el *script* del botón A, el primer paso es encontrar el personaje y posteriormente busca el nombre del objeto encontrado para averiguar si se trata del caballero o de la campesina. Una vez detectado el personaje ante el que se encuentra obtiene las partículas presentes en el objeto y procede de la forma adecuada.

En el caso del caballero, se debe de producir una modificación del daño de ataque. Para lograr este efecto se recupera el componente correspondiente al ataque y se aumenta el valor de la variable pública que hace referencia a las unidades de daño ejercidas por personaje. Durante la duración del efecto se produce la activación de las partículas del caballero, de esta forma se indica visualmente al jugador que el aumento de daño está presente. Tras concluir el período de mejora, las partículas se desactivan y comienza el tiempo de enfriamiento de la habilidad, tiempo en el cual no se puede utilizar la habilidad.

El efecto que tiene la habilidad sobre la campesina es el de aumentar su vida. En este caso, el *script* recupera el componente de vida del personaje y realiza un aumento en la vida actual del personaje. Durante la activación de la habilidad se activan las partículas de la campesina, indicando al jugador que se ha realizado la cura. Tras la desaparición de las partículas comienza el efecto de enfriamiento. Hay que aclarar que la campesina no podrá curarse cuando su vida llegue al máximo, esto hecho se controla en el fichero de su vida.

ScoreManager

El presente fichero es el encargado de contener una variable que almacene la puntuación de la partida y modifique el elemento de la interfaz que muestra ese valor. Este *script* esta añadido como componente al marcador de la interfaz. La funcionalidad que proporciona es la actualización continua del valor que aparece en el texto, la posibilidad de acceder a la variable y su correspondiente modificación, dado que se trata de una variable de tipo estático puede ser accedida fácilmente desde el resto de los *scripts*.

6.1.5. **Control de rondas**

El presente videojuego tiene un sistema de hordas que gestiona los enemigos que aparecen a lo largo de la partida. En el siguiente punto se explica toda la lógica detrás de sistema y se expone su implementación, desde la gestión de las rondas hasta el *spawn* de los enemigos.

El control de las rondas es gestionado por un *script* que contiene toda la implementación del sistema. Este fichero forma parte de los componentes de un objeto dedicado exclusivamente a realizar las tareas relacionadas con las rondas. A continuación, se presenta el componente mencionando explicando cada una de sus peculiaridades.

ControladorEnemigos

En este *script* se implementan las rondas descritas durante el apartado de diseño, además de controlar los elementos de interfaz correspondientes a la ronda actual y al fin de la partida. La llamada inicial *Start()* recupera la vida del personaje, ya que en caso de que muera se cesa la creación de enemigos.

Gracias a las corrutinas proporcionadas por Unity se han diseñado los distintos módulos del fichero. Este tipo de funciones nos da la posibilidad de realizar pausas en la ejecución, propiedad que se ha utilizado para sincronizar las distintas partes que intervienen en el sistema de rondas. Estas corrutinas se llaman en Unity mediante la función *StartCoroutine(corrutina)*. En el *script* podemos diferenciar tres funcionalidades principales que son las rondas, los cambios de letreros y los *spawns* de enemigos, estos se explican seguidamente.

El primer punto que vamos a tratar son las rondas, en la Imagen 25 podemos ver un ejemplo que se corresponde con una corrutina de una ronda. Las rondas empiezan llamando a la corrutina *InicioRonda()* y esperan el tiempo equivalente a la ejecución de esta. Una vez acabada la corrutina llamada, en la propia de la ronda se produce la creación de los enemigos utilizando las funciones *SpawnDiablo()* y *SpawnAranya()*. Cuando no se van a crear más enemigos se actualiza la variable *todosRespawn* a *true* indicando de esta forma que ha finalizado el tiempo de creación. Finalmente, se procede a esperar que se cumpla la condición de fin de ronda y se modifican las variables, aumentando *actualRound* en uno y poniendo *todosRespawn* a *false*, tras esto se llama a la corrutina de la siguiente ronda.

La segunda funcionalidad a tener en cuenta son las corrutinas modificadoras de elementos de interfaz. En concreto encontramos dos dentro del fichero, una para el letrero de ronda llamada *InicioRonda()* y otra para el de fin de partida con el nombre de *FinJuego()*. La primera de estas es llamada al inicio de cada ronda y muestra por pantalla la ronda actual que va a comenzar. La corrutina *FinJuego()* es llamada cuando termina la partida, ya sea por la muerte del jugador o por la superación de todas las rondas, mostrando en cada caso el texto correspondiente por pantalla.

```
IEnumerator Ronda1(int tiempo)
{
    StartCoroutine(InicioRonda());          // Llamada a la rutina que incia la ronda
    yield return new WaitForSeconds(5);     // Espara que sincroniza el tiempo de incio de ronda
    for (int i = 0; i < 10;)
    {
        SpawnDiablo(50,10,2,2, 10);         // Respawn diablo
        yield return new WaitForSeconds(tiempo);
        i++;
    }
    todosRespawn = true;                    // Todo los enemigos han spawneado

    while (!FinRonda())                     // Mientras no acabe la ronda esperamos
    {
        yield return new WaitForSeconds(3);
    }
    actualRound += 1;                       // Aumentamos la ronda actual

    todosRespawn = false;                   // Preparamos el valor para la siguiente ronda
    StartCoroutine(Ronda2());               // Llamada a la siguiente ronda

}
```

Imagen 25: Corrutina correspondiente a la primera ronda del videojuego

La ultima funcionalidad que encontramos son las dos funciones de *spawn*, sus nombre son *SpawnDiablo* y *SpawnAranya*. Ambas operan de forma que crean una instancia del enemigo con las características que se proporcionen a la función. De esta forma, podemos modificar los parámetros de vida, daño, velocidad de ataque, velocidad de movimiento y puntuación de cada enemigo, adaptando cada uno a la ronda correspondiente.

Por úlitmo, hay que mencionar la función *FinRonda()* que comprueba si todos los enemigos han sido creados, cosa que se hace a través de la variable *todosRespawn*, y si no queda ningún enemigo vivo. Esta función es utilizada para comprobar el fin de una ronda e iniciar la siguiente.

6.1.6. **Puntuación**

El sistema de puntuación del videojuego se basa en la acumulación de los puntos obtenidos por el jugador durante las partidas. Los enemigos están diseñados de forma que un diablo proporciona menos puntos que una araña, esto se debe a que los primeros aparecen con más frecuencia y es más fácil acabar con ellos. Por otro lado, el paso de las rondas aumenta la puntuación conseguida por enemigo eliminado dado que la dificultad aumenta con el avance de la partida.

Se han diseñado una serie de funciones para guardar y cargar un fichero que contenga la puntuación del jugador. Para poder realizar estas acciones se ha creado una clase llamada *PlayerData* la cual contiene una variable entera que se corresponde con la puntuación almacenada.

Para realizar el guardado de la puntuación se ha dispuesto de un objeto en la escena con el *script* llamado *GuardarPuntuacion.cs* como componente. Este fichero contiene la función *Save()* cuyo código podemos ver comentado en la Imagen 26. El primer paso realizado en la función es iniciar un codificador binario, con el que posteriormente guardaremos el fichero en binario. Seguidamente se comprueba si el fichero ya ha sido creado, en caso de no haber sido creado se crea en una ruta persistente. Tras esto se crea un objeto de la clase *PlayerData* y se le asigna a la variable *score* de este el valor de la puntuación obtenida en la partida. Finalmente, mediante el codificador guardamos binariamente el objeto en el fichero y cerramos este.

Por otra parte, puede darse el caso de que el fichero ya exista. En tal caso, se procede a abrir el archivo utilizando la función *Open()* y se recupera el objeto de la clase *PlayerData* con los puntos almacenados del jugador. A la variable del objeto se le añade la puntuación de la partida y se vuelve a guardar la puntuación como se ha comentado antes.

```
public void Save()
{
    BinaryFormatter bf = new BinaryFormatter();     // Iniciamos el codificador binario

    // Si no existe el fichero lo creamos y lo almacenamos en la ruta persistente
    if(!File.Exists(Application.persistentDataPath + "/playerInfo.dat"))
    {
        FileStream file = File.Create(Application.persistentDataPath + "/playerInfo.dat");

        PlayerData data = new PlayerData();     // Creamos un objeto PlayerData
        data.score = ScoreManager.score;        // Modificamos la puntuacion del objeto

        bf.Serialize(file, data);               // Serializamos el fichero con el objeto
        file.Close();                           // Cerramos el fichero
    }
    // Si ya existe el fichero lo abrimos
    else
    {
        FileStream file = File.Open(Application.persistentDataPath + "/playerInfo.dat", FileMode.Open);
        PlayerData data = (PlayerData)bf.Deserialize(file);  // Obtenemos los datos del fichero binarizado

        data.score += ScoreManager.score;                   // Añadimos la puntuacion a la del fichero
        bf.Serialize(file, data);                           // Serializamos el fichero con el objeto
        file.Close();                                       // Cerramos el fichero
    }
}
```

Imagen 26: Función *Save()* utilizada para guardar la puntuación en fichero

La función de cargado se realiza mediante la función *Load()* presente en el *script* llamado *CargarPuntuacion.cs*. La funcionalidad que implementa este fichero es igual a la que se ha explicado para el guardado cuando el archivo que almacena la puntuación ya existía. De esta forma, lo único que se diferencia con la explicación anterior es que el valor recuperado se pasa a un elemento de la interfaz de tipo texto y se muestra por pantalla. Esta función se utiliza en el menú principal para mostrarle al jugador cuantos puntos acumulados tiene.

6.2. Modo multijugador

En el siguiente apartado se procede a explicar cómo se ha desarrollado la modalidad multijugador, para ello partimos del trabajo realizado para el modo solitario. La distribución sigue el mismo patrón anterior, es decir, se hace una separación de las partes principales y se procede a su descripción.

Este modo se ha diseñado de forma que en una partida intervengan un total de dos jugadores, uno ejercerá el papel de host, es decir, será cliente y servidor simultáneamente, y el otro será un cliente conectado a este primero. Este hecho hay que tenerlo presente durante los distintos puntos para comprender bien las explicaciones.

6.2.1. Personajes

Para realizar el cambio de un modo a otro es necesario modificar varios aspectos de los personajes. Este cambio es bastante importante y significativo, esto se debe a que las modificaciones que seguidamente se expondrán dan al personaje propiedades de red y lo sitúan en la misma.

Componentes

Los personajes en el modo multijugador presentan los mismos componentes que tienen en el modo solitario adaptados para que tengan propiedades de red. Además, incluyen una serie de componentes adicionales necesarios para este modo que se describen a continuación.

- *Network Identity:* componente situado en el corazón de la API de alto nivel de red que proporciona Unity. Es el encargado de controlar la identidad única del personaje en la red y de esta forma logra que la red lo reconozca. En el caso de los jugadores la autoridad de este componente se sitúa en la parte local del jugador.

- *Network Transform:* componente que sincroniza el movimiento y la rotación del personaje sobre la red. Permite que otros jugadores vean los cambios realizados sobre el componente *transform* del personaje.

- *Network Animator:* componente que sincroniza las animaciones y los parámetros de los objetos que se encuentran en la red. Este componente recibe como parámetro un *animator* y en nuestro caso hace posible que los dos jugadores puedan ver las animaciones del otro.

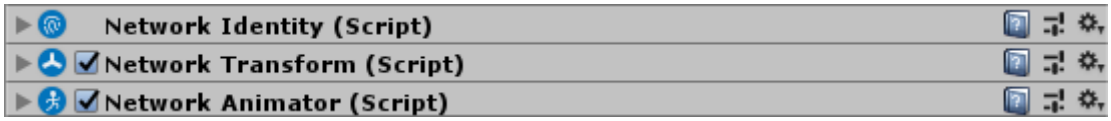

Imagen 27: Componentes adicionales de un personaje en el modo multijugador

Animación

Las animaciones se ven afectadas por el cambio, como se puede deducir de la adición del nuevo componente *Network Animator*. Los cambios no afectan a los estados de la máquina, que seguirán siendo los mismos y conteniendo sus respectivas animaciones. Sin embargo, los

parámetros y las transiciones entre los estados son sometidos a un cambio importante como podemos observar en la siguiente imagen.

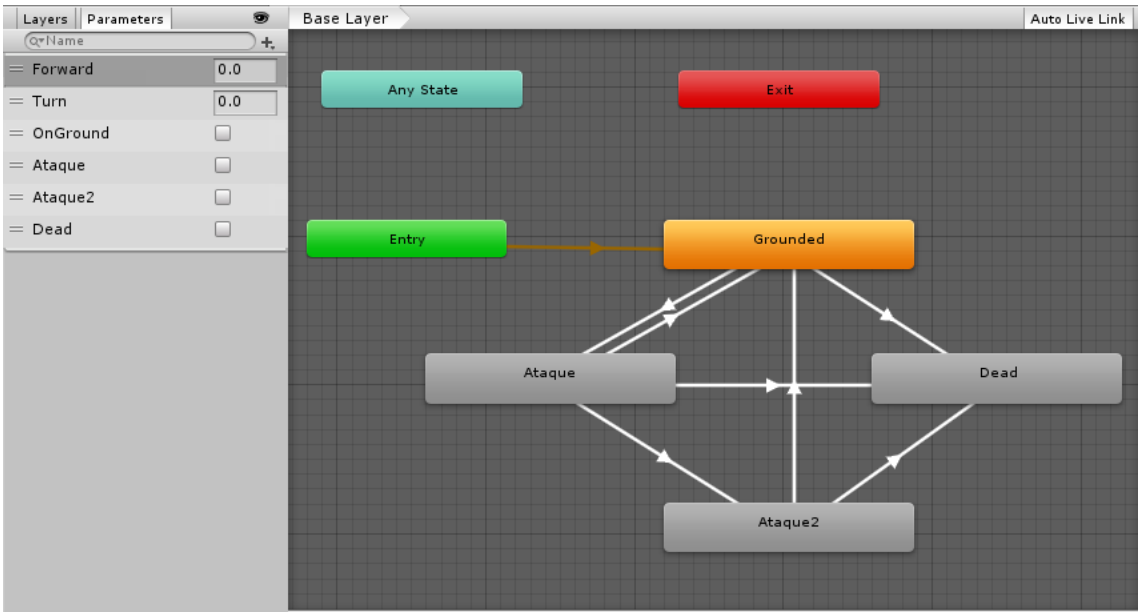

Imagen 28: Controlador de animaciones del caballero en el modo multijugador

Como se puede apreciar en la parte izquierda de la Imagen 28, todos los parámetros que antes eran de tipo *trigger* ahora son de tipo *bool*. Este cambio se debe a que el nuevo componente *Network Animator* no es capaz de sincronizar aquellos parámetros que son de tipo *trigger*. Este cambio, en el caso de los ataques, supone tener un parámetro por cada estado de ataque, esto es debido a la naturaleza de los *bool*, ya que no nos permite la realización de un combo con un solo parámetro, además de esta forma facilitamos el trabajo del componente.

El otro cambio importante lo vemos en el estado *Dead*, el cual recibe una transición desde cada uno de los estados, a diferencia de antes que solo recibía una del estado *Any State*. Esta disposición surge debido a que el componente *Network Animator* no reconoce el estado *Any State* y no es posible realizar la animación de muerte. Como se comprueba en la Imagen 28, la solución a este problema es la representación del estado *Any State* mediante el uso de transiciones, es decir, todos los estados tienen una transición hacia el estado *Dead* y se llegará a este independientemente del estado en el que se encuentre cuando el personaje muera.

Por último, hay que mencionar que los estados dedicados a las animaciones de ataque presentan un componente de tipo *script*. Este se encarga de devolver el valor del parámetro que ha activado el estado a *false*, realizando esta acción durante la llamada a la función *OnStateExit()* de la clase *StateMachineMonoBehaviour*, es decir, en la finalización del estado.

Scripts

La mayor modificación la encontramos en los *scripts* que formaban parte de los componentes de un personaje. Para el apartado de red, Unity proporciona la clase *NetworkBehaviour* la cual se ha utilizado para modificar los cinco *scripts* presentes en el modo solitario y crear uno más para solventar problemas de carga. Siguiendo el mismo formato utilizado para explicar *scripts* se procede a mostrar los cambios realizados en los *scripts*.

AnimacionPersonajeMulti

Como se ha comentado anteriormente, las animaciones han sufrido un gran cambio en los parámetros. Esta modificación no ha afectado a las animaciones de movimiento o muerte, sin embargo, las animaciones de los ataques requieren de un tratamiento especial ya que son activadas constantemente. Para explicar la implementación de las animaciones se utiliza como ejemplo el caso del caballero, cuyo código podemos ver en la Imagen 29.

El primer paso es diferenciar el tipo de personaje ante el que nos encontramos, ya que cada uno tiene un combo distinto. Una vez diferenciado el personaje, la primera condición a comprobar es el estado del primer parámetro, que en caso de tener su valor a *false* se procede a reiniciar el contador de golpes *ataque*, el cual contiene siguiente ataque a realizar. Este conjunto de instrucciones es utilizado para cubrir el caso en el que el jugador realiza el primer golpe pero no realiza el segundo, indicando que el siguiente ataque a realizar volverá a ser el primero. La segunda parte de la función está formada por una instrucción *switch* la cual recibe como parámetro el contador de golpes, activando la animación correspondiente en función del valor de este. El procedimiento para la campesina es el mismo con un estado más de ataque.

Gracias a esta implementación para el sistema de ataques, junto a los *scripts* que contienen los diferentes estados, es posible el funcionamiento de los ataques y combos en el modo multijugador.

```
public void Atacar()
{
    if (this.gameObject.name == "Multi_Guerrero(Clone)")    // Comprobamos el tipo de personaje
    {
        if (m_Animator.GetBool("Ataque") == false)          // Si el primer parametro es false,
        {                                                   // reiniciamos el contador de golpes
            ataque = 0;
        }
        switch (ataque)                                     // En funcion del contador, activamos la animacion
        {                                                   // correspondiente
            case 0:
                m_Animator.SetBool("Ataque", true);
                ataque++;
                break;
            case 1:
                m_Animator.SetBool("Ataque2", true);
                ataque = 0;
                break;
        }
    }
```

Imagen 29: Gestión de las animaciones de ataque en el modo multijugador

ControlPersonajeMulti

El único cambio realizado en este fichero, aparte de la extensión de la clase *NetworkBehaviour*, es la comprobación de que los cálculos realizados solo los llevara a cabo el jugador local. Esto se consigue comprobando la condición *isLocalPlayer* y haciendo un *return* en caso negativo.

InterfazPersonajeMulti

El presente *script* tiene la funcionalidad añadida de actualizar los parámetros presentes en los controladores de animación. Se ha decidido incluir este proceso en el archivo ya que está constantemente actualizándose, debido a que recoge los movimientos del personaje a través del *joystick*, y los parámetros de animación deben de sufrir el menor retraso posible.

VidaPersonajeMulti

En este fichero encontramos el primer problema relacionado con una variable que se debe compartir entre los jugadores. El valor de la vida de un personaje debe de estar sincronizado entre ambos jugadores, de forma que no se produzcan fallos como que un jugador observe la muerte del otro cuando a este le queda vida o a la inversa, es decir, que el otro jugador deba estar muerto y que el usuario actual lo visualice vivo. Este problema se solventa utilizando el tipo de variable sincronizada que proporciona Unity (*[SyncVar]*), cuyo valor será guardado en el servidor y mandado a los clientes, es decir, los jugadores.

Para que un cliente realice una modificación de una variable sincronizada es necesario utilizar un comando. Un comando es una función en Unity que se ejecuta en la parte del servidor, de este modo cuando los personajes sufren una modificación en su vida pueden modificar su valor en el servidor a través de un comando. En este *script* se realiza el uso de un comando para modificar el valor de la vida cuando la campesina utiliza su habilidad de curación, logrando de esta forma que ambos jugadores tengan el valor correcto en su partida.

AtaquePersonajeMulti

El funcionamiento del presente *script* es muy similar al del modo solitario, es decir, buscamos los enemigos que están en rango y procedemos a atacar. Las dos diferencias que encontramos son que podemos atacar a otro jugador y que las variables de vida son de tipo sincronizado. Así pues, en la detección del rango debemos diferenciar entre enemigos y el otro jugador, lo mismo sucede a la hora de atacar, en función de quien este en el rango realizaremos una llamada u otra. Por otra parte, las llamadas realizadas a las funciones de recibir daño se encuentran dentro de un comando, Imagen 30, existe uno para quitar vida a los enemigos y otro para realizar este efecto sobre el otro jugador. Con esto logramos sincronizar los golpes recibidos en la partida y que ambos jugadores tengan sincronizados los valores necesarios.

```
// Comando para realizar daño al otro personaje
[Command]
void CmdTakeDamage(GameObject otro)
{
    vidaOtroJugador = otro.GetComponent<VidaPersonajeMulti>();      // Obtenemos la vida del otro
    vidaOtroJugador.TakeDamage(attackDamage);                       // Realizamos daño al otro jugador
}
```

Imagen 30: Comando utilizado para atacar a otro jugador

Finalmente, en este fichero, al igual que hemos encontrado en el de vida, existe un comando encargado de modificar el daño. Este comando modifica la variable que contiene el dato de daño por golpe de un campeón, la cual es de tipo sincronizado. Su función es actualizar el daño del caballero cuando este utilice su habilidad especial, consiguiendo que en las partidas de ambos jugadores el daño sea el adecuado.

JugadorCargado

Se trata del *script* añadido para evitar un conflicto en el inicio de partida. Los personajes presentaban un problema durante su carga, este se producía por que los *scripts* y los hijos se ejecutaban antes de que el personaje estuviera cargado completamente arrojando mensajes de error. La solución implementada recoge los componentes y los objetos hijo que daban problemas, los cuales se encuentran desactivados durante la carga, para activarlos cuando se complete la carga del personaje. Posteriormente este *script* es destruido para evitar conflictos.

Objetos hijos

En el modo solitario los personajes contenían dos objetos, uno con la funcionalidad del rango de ataque y otro con los efectos de partículas propios de la habilidad especial. En el caso de la versión multijugador se añaden dos más, los cuales son la cámara y la interfaz, Imagen 31.

Este cambio viene provocado por la naturaleza del modo en red. Cada jugador tiene una instancia de la escena en su dispositivo, en la cual aparecerán los dos personajes que se hayan seleccionado. La cámara y la interfaz necesitan un objetivo sobre el que centrarse, en caso de estar situadas en la escena puede que seleccionen el personaje del otro jugador, cosa que supone un problema. Incluyendo estos objetos en el *prefab* del jugador, logramos establecer una relación entre ellos y ambos objetos se centran en el personaje adecuado.

Imagen 31: Objetos hijos en multijugador

Al igual que con el resto de los *scripts*, los que controlan la cámara y los elementos de la interfaz deben heredar de la clase *NetworkBehaviour*. El único cambio a destacar en los *scripts* de estos objetos lo encontramos en el fichero *BotonBMulti.cs* perteneciente al botón B de la interfaz. Se ha modificado la funcionalidad de este en las llamadas cuando se tenían que aplicar los cambios de las habilidades especiales. Ahora, en vez de acceder al valor y modificarlo, se produce una llamada al comando adecuado para que los cambios tengan la partida de ambos jugadores.

6.2.2. **Enemigos**

La transformación sufrida por los enemigos es muy similar a la de los personajes. A diferencia de los personajes, los enemigos tienen su acción localizada en la parte del servidor y desde el los jugadores reciben la información de estos.

Componentes

Los componentes adicionales de un enemigo son los mismos que se han añadido a un personaje, es decir, encontramos un *Network Identity*, un *Network Transform* y un *Network Animator*. La única diferencia la encontramos en el *Network Identity* el cual se ejecuta en la parte del servidor, mientras que en los personajes se ejecutan en la parte local del jugador.

Animación

Las animaciones sufren el mismo problema que los personajes por la adición del componente *Network Animator*. Es por ello que en el enemigo también podemos observar los cambios para solucionar los problemas, Imagen 32, es decir, la sustitución del estado *Any State* y el cambio de tipo de los parámetros. La única diferencia la encontramos en el estado *Idle*, que no está presente en los personajes, al cual llegaremos tras la muerte de los dos jugadores de la partida y se producirá la animación pertinente.

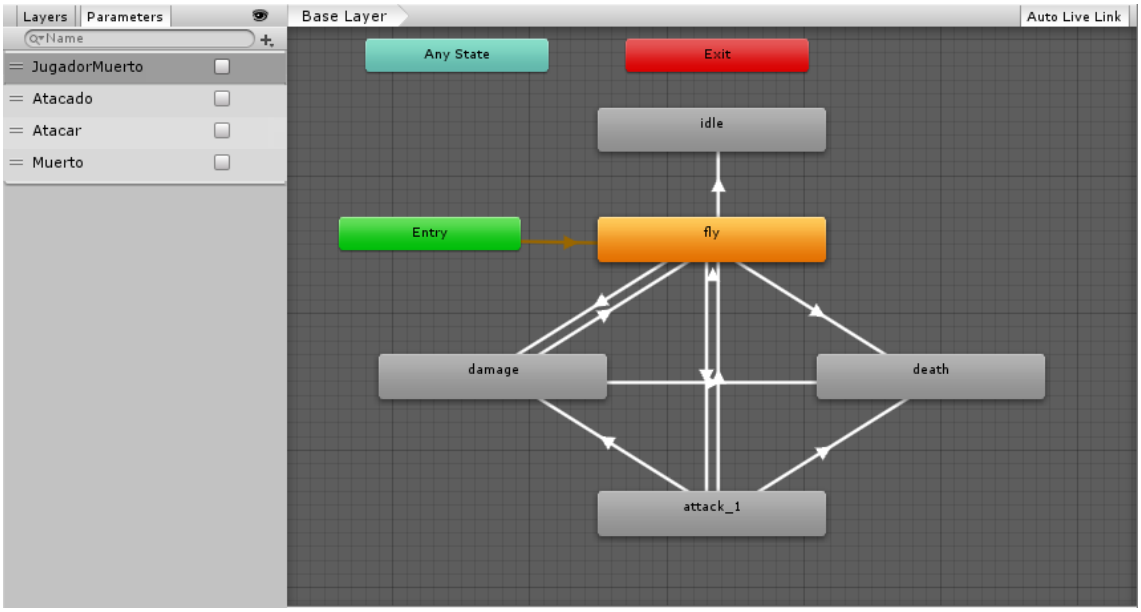

Imagen 32: Controlador de animaciones de un enemigo en el modo multijugador

Scripts

En los archivos de código de los enemigos solo encontramos modificaciones importantes en el movimiento. El ataque no requiere de ninguna sincronización ya que la vida de los personajes ya realiza esta función y por otra parte la vida de los enemigos, al igual que los personajes, tiene una variable sincronizada que almacena la vida, pero al crearse los enemigos en la parte del servidor no es necesario realizar más funciones.

MovimientoEnemigoMulti

En el presente *script* se han añadido varias funciones y modificaciones a la versión anterior del modo solitario. A continuación se explican las distintas funciones implementadas y posteriormente se explica el funcionamiento del *script*.

La primera función que vamos a mencionar se llama *BuscarJugadores()* y su función es localizar a los jugadores y almacenar los *GameObjects* correspondientes en un *array*. La segunda función implementada es *JugadoresMuertos()* y su objetivo es determinar si los jugadores han perdido toda su vida. Para comprobar esto se obtienen los componentes de vida de los personajes y se almacena el resultado en una variable de tipo *bool*, existe una variable por jugador. Finalmente, encontramos la función *DistanciaJugadores()* que partiendo de dos parámetros de tipo *transform*, uno por cada jugador, calcula cual de los jugadores está más cerca, estableciendo este como objetivo a perseguir por el enemigo.

En la función *Update()* encontramos las llamadas a las funciones descritas. El primer paso es localizar los objetos de los jugadores utilizando *BuscarJugadores()*. Una vez localizados comprobamos si alguno tiene vida, en caso afirmativo se procede a buscar aquel más cercano a la posición del enemigo marcándolo como objetivo a seguir. El seguimiento se ejecuta de la misma forma que en el modo solitario, utilizando el component *nav mesh agent* y la función *SetDestination()*.

Dejando de lado la funcionalidad del movimiento, en este *script* se ha incluido las instrucciones que actualizan los parámetros del animador, esto se debe a que se trata de un lugar ideal debido a las constantes actualizaciones. Las llamadas que realizan estas acciones son

SetParemeterAutoSend() y *GetParemeterAutoSend()*, estas actúan sobre un *network animator* y reciben como parámetro el índice correspondiente a un parámetro del controlador de animaciones.

6.2.3. *Lobby*

En este punto se expone el funcionamiento de la pieza que hace posible el modo multijugador. Se trata de un *prefab* que nos proporciona la interfaz de un *lobby* que nos permite elegir entre utilizar un sistema de servidores o que uno de los jugadores ejerza de *host*.

Para la implementación de nuestro videojuego hemos optado por la opción del *host*, es decir, un jugador deberá crear una partida y el otro unirse, como ya se ha explicado durante el apartado correspondiente de diseño.

El *script* que controla el *lobby* se llama *LobbyManager.cs* y se encarga de gestionar tanto los cambios en las interfaces como las conexiones a la red de los jugadores. A continuación se detallan las acciones más destacadas realizadas por este componente, tanto en la parte del cliente como en la del servidor.

Para el cliente se sobrescribe la función *OnClientConnect()*. El primer paso realizado dentro de la función es recuperar el personaje que ha seleccionado el jugador. Una vez realizado esto, se comprueba si existe un servidor conectado y se envía un mensaje con la información del personaje elegido.

Por parte del servidor se ejecutan más funciones. Las más visibles para el usuario son las que indican si el otro jugador está listo para jugar, cosa que se detecta a través de la pulsación del botón correspondiente, o la de cuenta atrás, activada cuando ambos jugadores están listos. Por otra parte, encontramos la función que asigna el *prefab* del personaje adecuado al jugador, el cual es recibido por el mensaje mandado del cliente y de esta forma el jugador pueda utilizar su selección.

Finalmente, encontramos una serie de funciones rutinarias encargadas de aspectos como la desconexión de un jugador, el cierre de sesión de una partida o los cambios de interfaz a medida que el usuario va interactuando con la aplicación.

6.2.4. **Control de rondas**

El sistema de rondas implementado en el modo multijugador sigue el mismo esquema realizado en el modo solitario, es decir, se emplean las corrutinas para definir las hordas y los cambios en las interfaces. El fichero utilizado para realizar toda la funcionalidad correspondiente tiene una serie de modificaciones que lo adaptan para el juego en red. Seguidamente se explican las peculiaridades de este.

ControladorEnemigoMulti

El presente *script* contiene la implementación del sistema de rondas, así como todas las funciones necesarias para crear los enemigos o manipular los elementos de tipo texto de la interfaz, como pueden ser la puntuación o el anuncio del final de partida. Durante la explicación resaltamos las diferencias que se han introducido en las funciones para su adaptación al modo multijugador.

El objeto que contiene el *script* tiene una identidad con autoridad situada en el servidor, esto implica que los cambios que se especifican en el fichero tienen efecto en el servidor y posteriormente son mandados a los clientes. Por este motivo, el inicio de las operaciones se produce en el servidor gracias al uso de la función *OnStartServer()*, donde se realiza la llamada que inicia la primera ronda.

El cambio más destacable lo encontramos en las corrutinas de las rondas, las cuales no pueden ser ejecutadas como se hacía anteriormente. Para poder ejecutar una corrutina y que esta tenga presencia en la red, se debe de utilizar un comando como intermediario, como podemos ver en la Imagen 33. El comando contiene la llamada a la corrutina utilizando la función *StartCoroutine(),* es por este hecho que por cada corrutina presente hay un comando que permite su llamada.

```
[Command]
void CmdRonda1()
{
    StartCoroutine(Ronda1(tiempoEntreSpawn));          // Llamada a la corrutina Ronda1()
}
```

Imagen 33: Comando con llamada a la corrutina Ronda1

Las funciones encargas de crear los enemigos tienen una pequeña modificación a la hora de hacer *spawn*. Tras realizar la instanciación y modificar los parámetros del enemigo, la aparición se realiza gracias a la función *NetworkSever.Spaw(enemigo).*

Finalmente, se incluye una función que comprueba si los jugadores han muerto, para parar la creación de enemigos en caso de que ambos hayan perdido su vida.

6.2.5. **Puntuación**

El sistema de puntuación es global y almacena los puntos de los enemigos eliminados por ambos jugadores, es decir, el marcador es compartido por ambos. El sistema de almacenamiento funciona de la misma forma que en el modo solitario y los puntos son almacenados en el dispositivo de forma acumulada.

En función del resultado de la partida, los jugadores reciben una cantidad distinta de puntos. En caso de acabar ambos vivos o muertos, la puntuación se divide entre los dos. Por el contrario, si solo un jugador queda vivo, este se queda una puntuación equivalente a multiplicar el valor total del marcador por 1.2.

7. Conclusiones

En lo que respecta al proyecto, al inicio de este nos propusimos la creación de un videojuego del género horda para la plataforma Android. Los juegos de este estilo en la plataforma de dispositivos móviles son escasos, este hecho reduce la competencia y hace que Android sea una buena decisión en términos de mercado.

Entre la variedad de motores gráficos del mercado disponibles al público, Unity3D ha sido seleccionado para el desarrollo de este proyecto por su sencillez, accesibilidad y versatilidad. Las diferentes funcionalidades que contiene y la forma en que estas se utilizan, hacen que el primer contacto con el desarrollo de un videojuego sea posible para alguien con conocimientos de programación y un poco de imaginación. Pese a tratarse de un motor sencillo de manejar, esto no implica que los resultados que proporcione sean peores que los que ofrecen sus competidores, ya que el correcto uso de las distintas herramientas disponibles en el motor hace posible la creación de obras magníficas.

El resultado del proyecto se debe al fruto obtenido de la implementación de los objetivos marcados al inicio y gracias a las utilidades proporcionadas por el motor, fue posible alcanzar las metas marcadas. Comenzando por el modo solitario, nos familiarizamos con la herramienta y creamos lo que sería la base del trabajo. Los personajes y los enemigos fueron creados añadiéndoles a sus modelos la funcionalidad pertinente, que una vez completada fue probada sobre el mapa de juego elaborado.

A la par que se desarrollaba el modo solitario, se creó el sistema de rondas del juego. En este proceso, las herramientas y estructuras proporcionadas por el motor fueron claves para diseñar a medida las diferentes rondas de una partida y controlar los enemigos que formarían parte de ellas.

Finalmente, alcanzamos a desarrollar el modo multijugador, completando así el último objetivo a mencionar. Para lograr implementar esta modalidad, se ha hecho uso de la utilidad presente en el motor para soportar este tipo de juegos. Dando propiedades de red a los diversos elementos del juego y utilizando los servidores gratuitos proporcionados por Unity se ha implementado este aspecto del videojuego.

7.1. Trabajo futuro

Los objetivos marcados al principio del proyecto han sido cumplidos satisfactoriamente y en este punto se puede pensar en posibles ampliaciones del trabajo a realizar en un futuro. A continuación, se exponen estas futuras posibles mejoras del trabajo.

Sistema de cosméticos

Una buena idea a implementar sería el diseño de un sistema de cosméticos que modifiquen la apariencia de los personajes durante la partida. Estos cosméticos cambiarían aspectos meramente visuales, sin proporcionar ventaja alguna al jugador, es decir, modificarían los efectos de las partículas y los trajes de los personajes. Loa cosméticos se podrían conseguir gastando los puntos acumulados de las partidas o pagando por ellos.

Variedad de personajes, enemigos y mapas

Otra posible mejora consistiría en la adición de nuevos personajes y enemigos con habilidades y aspectos diferentes, de forma que se aumente variedad que ofrece el juego. Los nuevos personajes podrían desbloquearse con el paso del juego o al igual que los cosméticos. Por otra parte, se añadiría la existencia de diferentes mapas con sus respectivos enemigos, lo cual podría llevar a la creación de una historia a través de los diferentes escenarios.

8. Bibliografía

[1] «El anuario del videojuego,» [En línea]. Available: http://www.aevi.org.es/web/wp-content/uploads/2018/06/AEVI_Anuario2017.pdf.

[2] «España, cuna del videojuego móvil,» [En línea]. Available: http://www.expansion.com/economia-digital/innovacion/2016/01/28/56aa4fd3ca4741e82e8b463c.html.

[3] B. R. Jimenez, «Andro4All,» [En línea]. Available: https://andro4all.com/2017/04/ventas-smartphones-febrero-2017-android-ios-espana.

[4] «Gears of War 4: Horde Mode,» [En línea]. Available: https://www.digitaltrends.com/gaming/gears-of-war-4-horde-mode-guide/.

[5] Infinity Ward, [En línea]. Available: https://www.callofduty.com/es/.

[6] «killingfloor2.com,» Tripwire Interactive , [En línea]. Available: http://www.killingfloor2.com.

[7] «Salmon Run - Inkipedia, the Splatoon wiki,» [En línea]. Available: https://splatoonwiki.org/wiki/Salmon_Run.

[8] «Unity - Editor,» [En línea]. Available: https://unity3d.com/es/unity/editor.

[9] «IDE de Visual Studio,» Microsoft, [En línea]. Available: https://visualstudio.microsoft.com/es/vs/.

[10] «Sublime Text - A sophisticated text editor for code, markup and prose,» [En línea]. Available: https://www.sublimetext.com.

[11] «About - Git,» [En línea]. Available: https://git-scm.com/about.

[12] «GitHub,» [En línea]. Available: https://github.com.

[13] « Free, open source, cross-platform audio software for multi-track recording and editing,» [En línea]. Available: https://www.audacityteam.org.

[14] «Unity - Manual: Scenes,» [En línea]. Available: https://docs.unity3d.com/Manual/CreatingScenes.html.

[15] «Unity - Manual: GameObjects,» [En línea]. Available: https://docs.unity3d.com/es/current/Manual/GameObjects.html.

[16] «Unity - Manual: Using Components,» [En línea]. Available: https://docs.unity3d.com/Manual/UsingComponents.html.

[17] «Unity - Manual: Transform,» [En línea]. Available: https://docs.unity3d.com/es/current/Manual/class-Transform.html.

[18] «Unity - Manual: Rigidbody,» [En línea]. Available: https://docs.unity3d.com/Manual/class-Rigidbody.html.

[19] «Unity - Manual: Colliders,» [En línea]. Available: https://docs.unity3d.com/es/current/Manual/CollidersOverview.html.

[20] «Unity - Manual: Animator,» [En línea]. Available:

https://docs.unity3d.com/Manual/class-Animator.html.

[21] «Unity - Manual: Audio Source,» [En línea]. Available: https://docs.unity3d.com/Manual/class-AudioSource.html.

[22] «Unity - Manual: Prefabs,» [En línea]. Available: https://docs.unity3d.com/es/current/Manual/Prefabs.html.

[23] «UnityScript's long ride off into the sunset - Unity Blog,» [En línea]. Available: https://blogs.unity3d.com/es/2017/08/11/unityscripts-long-ride-off-into-the-sunset/.

[24] «Unity - Scripting API: MonoBehaviour,» [En línea]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

[25] «Unity - Manual: Execution Order of Event Functions,» [En línea]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html.

[26] «Unity - Manual: La ventana de jerarquía,» Unity, [En línea]. Available: https://docs.unity3d.com/es/current/Manual/Hierarchy.html.

[27] «Unity - Manual: The inspector window,» Unity, [En línea]. Available: https://docs.unity3d.com/Manual/UsingTheInspector.html.

[28] «Unity - Manual: Ventana de lighting,» Unity, [En línea]. Available: https://docs.unity3d.com/es/current/Manual/GlobalIllumination.html.

[29] «Unity - Manual: The animator window,» Unity, [En línea]. Available: https://docs.unity3d.com/Manual/AnimatorWindow.html.

[30] «Unity - Manual: Profile window,» Unity, [En línea]. Available: https://docs.unity3d.com/Manual/ProfilerWindow.html.

[31] «Unity - Manual: Using the asset store,» Unity, [En línea]. Available: https://docs.unity3d.com/Manual/AssetStore.html.