Universitat Oberta de Catalunya

uoc.edu

Euphoria

Alejandro Díaz Sadoc
Máster Universitario en Diseño y Programación de Videojuegos
M7.462 – Trabajo Final de Máster

Heliodoro Tejedor Navarro
Joan Arnedo Moreno

06/06/2021

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Euphoria*
Nombre del autor:	*Alejandro Díaz Sadoc*
Nombre del consultor/a:	*Heliodoro Tejedor Navarro*
Nombre del PRA:	*Joan Arnedo Moreno*
Fecha de entrega (mm/aaaa):	06/2021
Titulación:	*Máster en Diseño y Programación de Videojuegos*
Área del Trabajo Final:	*M7.462 – Trabajo Final de Máster*
Idioma del trabajo:	*Castellano*
Palabras clave	*Videojuego, generación pseudoaleatoria, RPG*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El propósito de este proyecto se basa en comprender y analizar el proceso de creación de un videojuego desde el apartado creativo focalizándonos en el tema de diseño del juego y sus decisiones hasta el nivel de desarrollo. Se ha desarrollado un juego estilo “roguelike” donde los niveles se generan de manera aleatoria y hay que eliminar a un enemigo especial por nivel para ir avanzando en el juego. A través del desarrollo, se concluye lo importante que es la aleatoriedad en este tipo de juegos, así como tener en cuenta el nivel de dificultad del juego para no frustrar al jugador.

Abstract (in English, 250 words or less):

The purpose of this project is based on understanding and analyzing the process of creating a video game from the creative section focusing on the design of the game from the decisions that have been taken and the development level. A “roguelike” style game has been developed where the levels are randomly generated and you have to eliminate a special enemy per level to advance in the game. Through the development, it is concluded how important randomness is in this type of games, as well as taking into account the difficulty level of the game to not frustrate the player.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

El proyecto actual cubre el tema de la generación aleatoria o procedural en los videojuegos, específicamente en los videojuegos del subgénero "*roguelike*", buscando el resultado de obtener un videojuego con una generación aleatoria de los diferentes niveles a través de la elección aleatoria de diferentes habitaciones diseñadas e implementadas previamente, así como, de la generación de enemigos de forma aleatoria dentro de dichas habitaciones y por último la generación aleatoria de objetos una vez estos enemigos van siendo eliminados todo esto para comprobar el proceso de crear un videojuego de este tipo desde cero.

1.2 Objetivos del Trabajo

En esta sección se detallan los diferentes objetivos que se planean en un principio a desarrollar para completar el desarrollo del proyecto:

- Diseño y creación de plantilla de niveles
- Diseño y creación de la historia
- Creación de los diferentes niveles
- Creación de los diferentes elementos
- Implementación aleatoriedad de la generación de elementos
- Implementación selección aleatoria/generación procedural de los niveles
- Creación e implementación del personaje principal
- Creación e implementación de los enemigos
- Creación IA/Patrones de los enemigos
- Implementación de la historia
- Corrección bugs/errores.

1.3 Enfoque y método seguido

En el desarrollo del proyecto se ha optado por el uso de una metodología ágil ya que se adapta mejor a la experiencia de trabajo y al cumplimiento de objetivos planteados. Ya que envuelve un enfoque para la toma de decisiones en los proyectos de software, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto. Así se consigue que el trabajo sea realizado mediante equipos autoorganizados y multidisciplinarios. Consiguiendo así una respuesta rápida a valoraciones o cambios especificados sobre el proyecto.

El objetivo principal de este proyecto es el de desarrollar un producto totalmente nuevo, es decir, desarrollar un videojuego completamente desde cero.

1.4 Planificación del Trabajo

A continuación, se especifican las diferentes tareas que se van a realizar que se pueden considerar como los objetivos que se planean desarrollar y una estimación de tiempo a través también de un Diagrama de Gantt (véase Figura 1).

- Diseño y creación de plantilla de niveles: 1 semana
- Diseño y creación de la historia: 5 días
- Creación de los diferentes niveles: 1 semana
- Creación de los diferentes elementos: 3 días
- Implementación aleatoriedad de la generación de elementos: 2 días
- Implementación selección aleatoria/generación procedural de los niveles: 10 días
- Creación e implementación del personaje principal: 7 días
- Creación e implementación de los enemigos: 15 días
- Creación IA/Patrones de los enemigos: 15 días
- Implementación de la historia: 6 días
- Corrección bugs/errores: 15 días

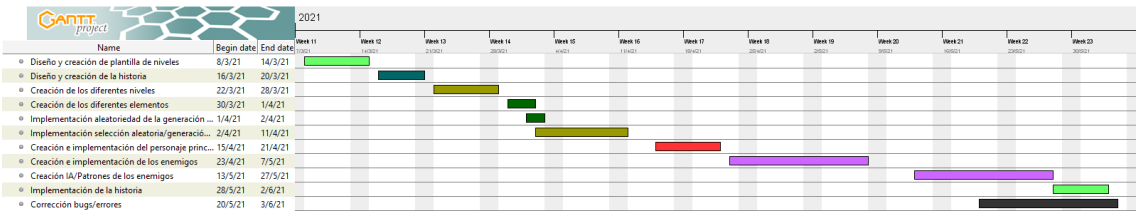

Figura 1 Diagrama de Gantt del proyecto

1.5 Breve sumario de productos obtenidos

El resultado obtenido del producto se considera una versión parcial de lo que se podría considerar como videojuego real y acabado muchos más elementos, pero en dicha versión se puede ver los elementos más importantes y los que se podrían considerar como elementos base del juego desarrollados.

Se han desarrollado de manera completa la generación aleatoria de los niveles con la elección aleatoria de las diferentes habitaciones además del comportamiento de los diferentes enemigos y la generación de estos en las habitaciones generadas aleatoriamente.

Además, se ha obtenido la interfaz de usuario para el jugador y la interacción por parte del jugador con los diferentes enemigos, objetos y

con lo necesario para ir progresando en el videojuego y pasar a nuevos niveles.

1.6 Breve descripción de los otros capítulos de la memoria

1. Introducción: Breve introducción del proyecto y sus objetivos además de la planificación de desarrollo de este.
2. Estado del arte: Breve revisión sobre el estado actual del género explicando las diferentes características de algunos de los juegos más recientes de este género y revisión sobre lo diferentes motores de videojuegos utilizados para desarrollar este tipo de videojuegos.
3. Análisis y requisitos del sistema: Descripción de los requisitos software con los que cuenta el proyecto y la interacción de los diferentes componentes que componen el proyecto.
4. Diseño: Explicación sobre las decisiones de diseños que se han tomado a lo largo del desarrollo del proyecto.
5. Implementación: Explicación de todo el desarrollo de los diferentes objetivos del proyecto.
6. Manual de usuario: Explicación de guía del usuario sobre cómo se ejecuta el proyecto y se juega al videojuego desarrollado.
7. Conclusiones: Explicación de las conclusiones que se han obtenido tras el desarrollo del proyecto y del posible trabajo futuro a elaborar.

2. Estado del arte

A continuación, se va a realizar una revisión sobre el género principal del videojuego desarrollado, el cual trata de un videojuego de género "*roguelike*". Este genero es un subgénero dentro del mundo de los videojuegos conocidos como RPGs y se basa en el uso de la aleatoriedad y la repetición como base de los videojuegos donde el objetivo general del jugador es explorar una mazmorra para derrotar a un *boss* y así poder continuar con el juego, donde la importancia de estas mazmorras es que precisamente se generan de manera aleatoria y de contar con una muerte permanente del personaje, es decir, una vez el jugador muere se tiene que volver a empezar todo desde cero.

En 1980 se publicó el videojuego **Rogue**, el cual se trataba de un videojuego en el cual los jugadores controlaban a un héroe que se adentraba en unas mazmorras y se enfrentaba a diferentes enemigos para encontrar un amuleto. Una vez el jugador era eliminado, la partida se acababa y el jugador tenía que volver a empezar desde cero con un nuevo mapa generado aleatoriamente con nuevos enemigos. Este juego es en el cual el género "*roguelike*" se basa, como se puede comprobar en la propia palabra "*roguelike*" que lleva incluido **Rogue** en sí misma, y al traducirla al castellano quedaría como "parecido a Rogue".

En la actualidad existen grandes ejemplos de videojuegos estilo "*roguelike*" que son bastante famosos y son considerados como grandes videojuegos por parte de los usuarios mediante las diferentes críticas y notas que les otorgan.

El ejemplo por excelencia en la actualidad cuando hablamos sobre videojuegos "*roguelike*" es el videojuego "***The Binding of Isaac***" [11], el cual se trata de un videojuego indie diseñado por **Edmund McMillen** y **Florian Himsl**. En este juego se controla a un personaje a través de una mazmorra generada procedimentalmente y consiste en ir derrotando a los monstruos en combates a tiempo real mientras recogen objetos y potenciadores para derrotar a los enemigos tipo *boss* y, por último, como último *boss*, a la madre de **Isaac**. Este juego obtuvo una adaptación en 2014, año donde el propio **McMillen** informó que ya se habían vendido más de 3 millones de copias de este y se puede considerar como el juego que más contribuyó a la hora de generar interés por el género "*roguelike*".

Además de **The Binding of Isaac**, en los últimos años se cuenta con otro gran ejemplo de un gran videojuego del estilo "*roguelike*" como es el videojuego **Hades**. **Hades** [10] es un videojuego desarrollado y publicado por **Supergiant Games** lanzado oficialmente en 2020, y en este juego, el jugador controla a **Zagreus**(hijo de **Hades**) mediante diferentes habitaciones generadas aleatoriamente pobladas de enemigos y recompensas, con la diferencia en la que en el caso de que el jugador muera, este puede usar las diferentes recompensas ganadas

para mejorar al personaje de manera posterior para mejorar las posibilidades de escapar en los intentos posteriores a este.

Y como último, ejemplo, este mismo año, ha sido lanzado oficialmente el juego **Loop Hero** [9], juego de estilo "*roguelike*" desarrollado por **Four Quarters** el cual tiene lugar en un mundo generado aleatoriamente en el cual el jugador controla dicho mundo colocando cartas en lugar de controlar directamente a un personaje. Es decir, el juego comienza en un mundo generado aleatoriamente el cual, el personaje va a recorrer cada día del propio juego. Durante el recorrido de este mundo, el personaje se enfrenta a diferentes enemigos, y tras vencerlos el jugador obtiene cartas como recompensa, de esta manera interactúa el jugador con el personaje, es decir, mediante las cartas que son las encargadas de tener efectos diferentes sobre el propio personaje (como aumentar la velocidad, por ejemplo).

Como se han podido comprobar con estos tres ejemplos, todos tienen mecánicas muy diferentes entre ellos mismos, pero aún así todos comparten la misma base, es decir, comparten la generación aleatoria de los diferentes niveles/mundo con respecto a los enemigos a los que se enfrentan, las recompensas que reciben, etc.

2.1 Revisión tecnológica de los juegos "*roguelike*"

Como se puede comprobar, la mayoría de los videojuegos nombrados anteriormente han sido desarrollados por desarrolladoras *indies* (independiente), por lo tanto, son videojuegos considerados como *indies*. Estos videojuegos son caracterizados debido a que son creados normalmente por un equipo de desarrollo pequeño y sin el apoyo financiero y técnico de un gran editor de videojuegos, a diferencia de los videojuegos considerados como videojuegos triple A ("AAA").

Teniendo esto último en cuenta, es bastante común pensar que para los desarrolladores en este caso sería más cómodo y rápido utilizar un motor de videojuegos ya creado para desarrollar sus propios videojuegos, pero en algunos casos esto no es así. En algunos juegos los desarrollados han optado por utilizar sus propios motores de videojuegos desarrollados por ellos mismos en lugar de confiar en los motores de videojuegos más conocidos en el mercado. Esta decisión puede verse basada en el hecho de que en estos juegos no es tan importante la potencia gráfica como lo son en otros juegos triple A, es decir, en estos juegos se premia más la aleatoriedad y la jugabilidad junto con el arte que la propia potencia del motor de videojuegos.

Esta podría ser una de las razones por las cuales los videojuegos "*roguelike*" no necesitan de un gran motor de videojuegos para desarrollarse y el desarrollo de estos se basa más en una gran aleatoriedad en la generación del mundo y los enemigos, una jugabilidad con aspectos e idea tanto innovadoras como interesantes para mantener a los jugadores con ganas de jugar varias veces al juego

para encontrarse con situaciones diferentes en cada momento y un arte normalmente “bonito” o con un gran trabajo detrás, en lugar de centrarse en el uso de grande gráficos realistas.

Estos juegos son un ejemplo de que no son necesarios unos grandes gráficos ni tanta potencia para ejecutar estos gráficos a la hora de desarrollar grandes videojuegos, y se han mencionado grandes videojuegos que consiguen esto último, los cuales son: **The Binding of Isaac, Hades** y **Loop Hero.**

3. Análisis y requisitos del sistema

En este capítulo se describirán los requisitos con los que cuenta el proyecto de ***Euphoria***, analizando y especificando todos los componentes software que componen el proyecto en su totalidad. Además, se expondrán los diferentes casos de uso de interacción por parte del jugador con el proyecto, así como, la interacción de los componentes del proyecto en sí.

3.1 Requisitos del sistema

A continuación, se van a especificar los diferentes requisitos funcionales software de nuestro sistema, es decir, aquellos requisitos con los que tiene que contar el sistema/proyecto para cumplir el objetivo de nuestro trabajo.

3.1.1 Software

Los requisitos software de nuestro sistema son:

- El juego debe de ser capaz de generar el diferente nivel actual a través de la aleatoriedad en la selección de diferentes habitaciones previamente diseñadas.
- Esta generación se debe realizar de forma finita, es decir, el nivel se debe de encontrar cerrado en todos sus límites.
- Una vez realizada toda la generación, las diferentes habitaciones del nivel tienen que encontrarse de alguna manera conectadas entre sí, para conseguir que el jugador pueda ir cambiando de habitaciones de forma coherente.
- El jugador debe de ser capaz de cambiar de habitaciones de una forma coherente y sin ningún problema.
- A través de la generación de las diferentes habitaciones, dicha generación deberá de ser capaz de generar de forma aleatoria los diferentes enemigos que se van a encontrar en la habitación para oponer resistencia al jugador.
- El sistema deberá de ser capaz de detectar en que habitación se encuentra el jugador actualmente, para habilitar el comportamiento de los diferentes enemigos que se encuentran en dicha habitación conforme al jugador.
- Los enemigos deberán de ser capaces de detectar al jugador y realizar su comportamiento de manera correcta, ya sea disparar al jugador o perseguirlo.
- El sistema deberá de ser capaz de ir actualizando la interfaz de usuario (UI) del juego, para cada cambio relevante ya sea modificación en la salud actual del personaje o cambio de habitación del jugador con respecto al minimapa de la UI.
- El sistema deberá de ser capaz de otorgar al jugador mejoras a través de la eliminación de los diferentes enemigos comunes que se encuentran en las diferentes habitaciones.

3.2 Diagrama del sistema

A continuación, se muestra un diagrama que contienen los diferentes "componentes" que forman el sistema y la comunicación entre estos, así como sus tareas a realizar. El componente principal de todos los que forman el sistema de una manera desglosada, es el componente de Generación de nivel/*GameManager* ya que entre ambos se encargan del funcionamiento principal del juego:

- Generar el nivel a través de la aleatoriedad en la elección de las diferentes habitaciones que componen el nivel.
- Generar los diferentes enemigos en las diferentes habitaciones que componen el nivel.
- Enlazar las diferentes habitaciones para permitir una correcta navegación al jugador por todo el nivel.
- Gestionar los diferentes niveles que va superando el jugador.
- Actualizar los elementos correspondientes de la UI por parte del minimapa.

Además de este componente de generación de nivel. Nos encontramos la parte que se encarga de gestionar el movimiento del jugador y las estadísticas del personaje dentro del juego (vida, etc…) y el comportamiento de los enemigos dentro de las habitaciones, así como, su comportamiento con respecto al jugador. Además de estos dos, nos encontramos un último componente principal que es el encargado de gestionar la UI del juego, actualizando la UI con respecto a los diferentes componentes. En la Figura 2 se puede ver el diagrama del sistema.

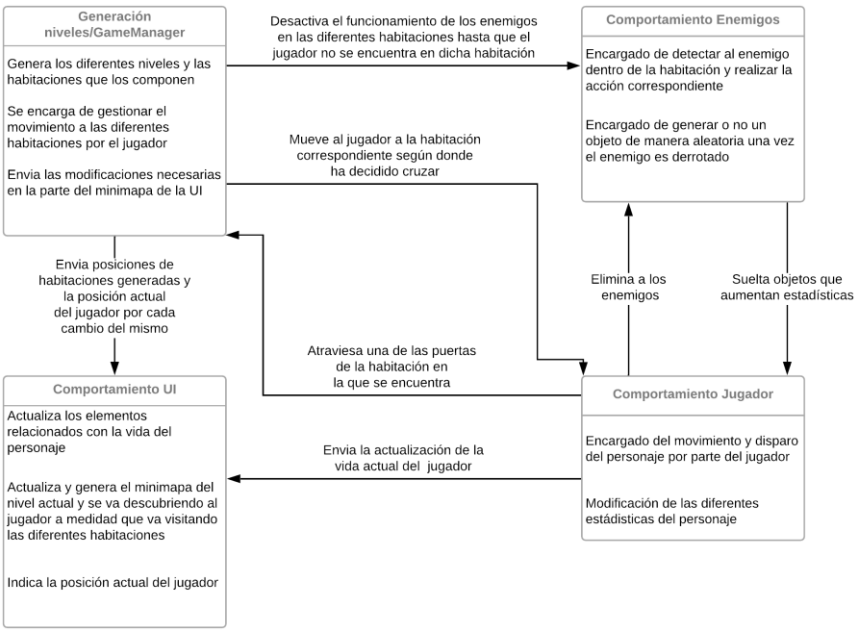

Figura 2 Diagrama del sistema

3.3 Actores

En los actores correspondiente, en este caso, solo contamos con el propio usuario que ejecuta el juego, es decir, el propio jugador en este caso y el propio sistema en sí, que lanza la respuesta en consecuencia a las diferentes acciones que realice el jugador.

3.4 Casos de uso

A continuación, se van a exponer los diferentes casos de uso más importantes que se han identificado en la relación entre el jugador y el sistema de componentes ya explicado anteriormente:

- Jugador cambia de habitación
- Jugador cambia de nivel
- Jugador elimina a un enemigo
- Jugador es eliminado
- Jugador recoge objeto

3.5 Descripción de los casos de uso

En esta sección se van a exponer las diferentes descripciones de los casos de uso definidos en la sección 3.4 Casos de uso:

Descripción del caso de uso "Jugador cambia de habitación"

- Nivel: Acción
- Actor principal: Jugador
- Actor secundario: Sistema
- Descripción: Detecta la intención del jugador de cambiar de habitación en el nivel actual
- Precondiciones:
  - El nivel ya debe de haber sido generado completamente.
  - El minimapa debe de haber sido generado completamente.
- Postcondiciones: El sistema detecta la colisión por parte del jugador con una de las puertas de la habitación actual y mueve al jugador a la habitación correspondiente que se encuentra enlazada por dicha puerta con respecto a la habitación actual.

  - **Escenario Principal (Éxito):** Cambiar de habitación.

- **Escenario Principal:**
  1. El jugador colisiona con una de las salidas de la habitación actual.
  2. Unity detecta esta colisión y es manejada por los diferentes *colliders* que se encuentran en la habitación.
  3. El sistema realiza un cálculo con respecto a las coordenadas del jugador para poder distinguir el *collider* con el cual se ha interactuado para conocer así la dirección en la que se desea mover el jugador.

4. El sistema desactiva la habitación actual en la que se encuentra el jugador, así como, a los enemigos y activa la nueva habitación.
5. Al activar la nueva habitación todos los enemigos de la nueva habitación son activados y pueden atacar al jugador.
6. El sistema mueve al jugador a la nueva habitación.
7. El sistema mueve la cámara de juego hacia la nueva habitación
8. El sistema notifica a la UI de este cambio de habitación para que el minimapa se vea afectado.
9. El sistema de UI calcula el cambio de habitación realizado con respecto al nivel y al minimapa previamente generado.
10. El sistema de UI actualiza la interfaz de usuario y modifica la posición del jugador en el minimapa.
    a. La habitación no había sido visitada anteriormente por lo que se ha descubierto una nueva habitación en la parte del minimapa y debe de aparecer como visible.
    b. La habitación se había descubierto previamente y esto no modificada su estado con respecto al minimapa.

Descripción del caso de uso "Jugador cambia de nivel"

- Nivel: Acción
- Actor principal: Jugador
- Actor secundario: Sistema
- Descripción: Detecta la intención del jugador de cambiar de nivel
- Precondiciones:
    - El nivel ya debe de haber sido generado completamente.
    - El *boss* actual del nivel debe de haber sido derrotado.
- Postcondiciones: El sistema detecta la colisión por parte del jugador con el hueco de salida de la habitación en la que se encuentra el *boos* del nivel y cambia el nivel actual y genera uno nuevo.

    - **Escenario Principal (Éxito):** Cambiar de nivel.

- **Escenario Principal:**
    1. El jugador encuentra la habitación en la que se encuentra el *boss* del nivel.
    2. El jugador elimina al *boss* de dicho nivel.
    3. El sistema muestra el hueco de salida en el centro de la habitación actual (la habitación del *boss*).
    4. El jugador colisiona con el hueco de salida.
    5. Unity detecta esta colisión y es manejada de forma especial al tratarse de la salida del nivel.
    6. El sistema realiza un borrado de todo lo relacionado con la generación del nivel actual.
    7. El sistema UI realiza un borrado de todo lo relacionado con la generación del minamap actual.
    8. El sistema empieza a generar el nuevo nivel de forma aleatoria.
    9. Durante la generación se genera además el minimapa con respecto al sistema de UI.

10. El jugador es desplazado nuevamente a la nueva habitación actual del nivel.
11. El sistema de UI actualiza la posición del jugador en el minimapa que se acaba de generar en el nuevo nivel.
12. El sistema de UI comprueba que el minimapa actual no exceda los límites establecidos para la UI y no quede parte del minimapa completamente invisible.

Descripción del caso de uso "Jugador elimina a enemigo"

- Nivel: Acción
- Actor principal: Jugador
- Actor secundario: Sistema
- Descripción: El jugador a través de varios disparos acaba eliminando al enemigo tras varias colisiones entre las balas y el enemigo.
- Precondiciones:
    - Los enemigos de la habitación actual deben de haber sido generados.
    - El enemigo debe de haber recibido el disparo o disparos anteriormente.
- Postcondiciones: El sistema detecta la colisión del enemigo con la bala dispara anteriormente por el jugador y comprueba si el daño de la bala.

    - **Escenario Principal (Éxito):** Jugador elimina a enemigo

- **Escenario Principal:**
    1. El jugador realiza un disparo en dirección al enemigo.
    2. El disparo impacta contra el enemigo.
    3. Unity detecta la colisión del disparo con el enemigo.
    4. El sistema comprueba que el disparo ha sido realizado por parte del jugador para distinguir entre los disparos realizados por los enemigos y el jugador.
    5. El sistema comprueba que el daño causado por el disparo es mayor o igual que la cantidad de salud restante del enemigo.
    6. El sistema destruye el disparo
    7. El sistema destruye al enemigo.
    8. El sistema decide aleatoriamente si el enemigo soltará un objeto para el jugador
        a. La probabilidad no ha sido superada por lo tanto no se soltará ningún objeto para el jugador
        b. La probabilidad ha sido superada y se soltará un objeto de manera aleatoria que aumente las estadísticas del jugador.

Descripción del caso de uso "Jugador es eliminado"

- Nivel: Acción
- Actor principal: Sistema
- Actor secundario: Jugador

- Descripción: Detecta la colisión de un disparo o ataque por parte del enemigo hacia el jugador.
- Precondiciones:
    - El enemigo debe de haber realizado un ataque o disparo.
    - El jugador debe de haber sufrido daño previamente.
- Postcondiciones: El sistema detecta la colisión por parte de un disparo o ataque de los enemigos hacia el jugador, comprueba que el daño causado es mayor que la vida actual del jugador y elimina al jugador llevándolo a la escena de juego terminado.

    - **Escenario Principal (Éxito):** Jugador es eliminado.

- **Escenario Principal:**
    1. El enemigo realiza el ataque o disparo en dirección al jugador.
    2. El sistema detecta la colisión de dicho ataque o disparo con el jugador.
    3. El sistema le ajusta la vida al jugador con respecto al daño causado por parte del disparo o ataque.
    4. En caso de que se trate de un disparo, el sistema destruye el disparo.
    5. El sistema comprueba que el daño que ha causado el ataque o el disparo es mayor a la cantidad de vida actual del jugador.
    6. El sistema verifica que la salud es 0 o menor que 0 y destruye el objeto del jugador.
    7. Tras destruir el objeto, el sistema cambia a la escena de juego terminado.
    8. Una vez cargada dicha escena, se le ofrece las posibilidades al jugador de jugar de nuevo o salir del juego.

Descripción del caso de uso "Jugador recoge objeto"

- Nivel: Acción
- Actor principal: Jugador
- Actor secundario: Sistema
- Descripción: Detecta la colisión de uno de los objetos soltados por la muerte de un enemigo con el jugador.
- Precondiciones:
    - El nivel ya debe de haber sido generado completamente.
    - Los enemigos en cada habitación deben de haberse generado completamente.
    - El enemigo relacionado con dicho objeto debe de haber sido eliminado por parte del jugador.
- Postcondiciones: El sistema detecta la colisión por parte del jugador con el objeto soltado por la eliminación de uno de los enemigos y si es posible se le añaden las estadísticas de dicho objeto al jugador.

    - **Escenario Principal (Éxito):** Jugador aumenta sus estadísticas tras recoger objeto.

- **Escenario Principal:**

1. El jugador colisiona con uno de los objetos predefinidos y soltados por la eliminación de un enemigo.
2. Unity detecta esta colisión y se maneja por parte del comportamiento de los objetos.
3. El sistema comprueba si es posible añadirle las estadísticas del objeto con el que se ha detectado la colisión al jugador.
    a. En caso de poder añadir dichas estadísticas, el jugador ve aumentada la estadística correspondiente y el objeto es eliminado por parte del sistema.
    b. En caso de no poder añadir dichas estadísticas, el jugador no ve incrementadas sus estadísticas y el objeto no es eliminado por parte del sistema.

4. Diseño

En este capítulo se especifican todas las decisiones de diseños que se han tomado a lo largo del desarrollo del videojuego, así como, también se especifican las razones por las que se ha decidido tomar esas decisiones de diseño.

4.1 Estudios de alternativa y viabilidad

El diseño de nuestro videojuego se basa principalmente y como punto fuerte en la generación automática y aleatoria de los diferentes niveles que componen el juego conforme el jugador va avanzando en el mismo.

Tendiendo esto último en cuenta, para el desarrollo del videojuego se estudió la posibilidad de realizarlo en diferentes motores de videojuegos, llegando a la decisión de optar por Unity3D como el motor de videojuegos a usar para desarrollar el proyecto. En esta sección se explicarán los siguientes engines/motores de videojuegos:

- **Unreal Engine**
- **Godot Engine**
- **Unity3D**

4.1.1 Unreal Engine

Unreal Engine [7] es un motor de videojuego desarrollado por **Epic Games** el cual es compatible con una amplia gama de plataforma, y los juegos desarrollados en **Unreal** pueden ser exportados a diferentes plataformas. **Unreal** está escrito en C++, por lo que el lenguaje usado para desarrollar los scripts se trata de C++. Además de poder desarrollar los scripts a través de lenguaje, **Unreal** aporta la posibilidad de crear los scripts a través de *blueprints*, el cual se trata de un sistema de *scripting* visual que permite desarrollar rápidamente la lógica del juego sin utilizar código, lo que reduce la división entre artistas técnicos, diseñadores y programadores.

4.1.2 Godot Engine

Godot [8] es un motor de videojuegos enfocado en el desarrollo de videojuegos tanto 2D como 3D, de código abierto baja la licencia de MIT y puede exportar los diferentes juegos que se realizan en **Godot** a las siguientes plataformas:

- PC
- Móvil
- Navegador web

Una de las ventajas que ofrece Godot es la flexibilidad en los lenguajes que se pueden realizar los juegos, pudiéndose utilizar lenguajes como **C++**, **C#** y otros tipos de lenguaje como **Rust**, **Nim**, y **D**. Además, cuenta con un tipo de lenguaje especifico y creado para este motor, llamado ***GDScript***, el cual es un lenguaje de alto nivel y dinámicamente tipado bastante similar a **Python**, y se encuentra optimizado para la arquitectura basada de las escenas de **Godot**.

4.1.3 Unity3D

Unity [6] es un motor de videojuegos desarrollado por ***Unity Technologies***, el cual puede ser usado para desarrollar juegos 3D, 2D, de realidad virtual y de realidad aumentada, y es una tecnología que ha sido usada y adaptada para otros ámbitos más allá de la creación de videojuegos.

El lenguaje utilizado para crear los scripts utilizados en este motor de videojuegos es **C#**. En este caso, para el desarrollo del Trabajo Final de Máster, se ha optado por esta opción de **Unity**, debido a la experiencia que se obtiene tras los varios desarrollos en el mismo máster, además de la cantidad de *assets* y modelos que existen por parte de la comunidad de **Unity** para el desarrollo de diferentes juegos. Además de **Unity** se va a usar en conjunto o bien el IDE de **Visual Studio** para tener una integración completa de **Unity** con el entorno de desarrollo o el editor de **Visual Studio Code** ambas opciones para desarrollar los scripts en **C#**.

4.1.3.1 Requisitos Unity3D

A continuación, se especifican los diferentes requisitos necesarios para poder ejecutar Unity3D completamente [17]:

- Sistemas operativos: Windows 7 SP1+, 8, 10, solo versiones 64-bit; Mac OS X 10.12+; Ubuntu 16.04, 18.04, and CentOS 7.
- Tarjeta gráfica (GPU): Tarjeta de video con capacidad para DX10 (shader modelo 4.0).

4.2 Arquitectura del sistema

A continuación, se especifican las diferentes funciones que realizan cada uno de los componentes que se pueden considerar como individuales que trabajan en conjunto para conformar la arquitectura del sistema, y ejecutar la lógica completa del videojuego.

4.2.1 Generador de niveles

El generador de niveles es el encargado de la generación aleatoria de los diferentes niveles del juego. Esta aleatoriedad está basada en la

elección aleatoria de habitaciones previamente diseñada e implementadas a través de objetos ***prefabs***. Esta aleatoriedad se ve afectada por la elección del tipo de habitación a elegir, es decir, este componente se relaciona intrínsicamente con el componente de generador de habitaciones, a través del cual, se especifica el tipo de habitación que se requiere junto a la habitación previamente generada (habitación con puerta a la izquierda, derecha, etc.). Además, el generador de niveles es el encargado de especificar la habitación en la que se va a encontrar el ***boss*** del nivel actualmente generado. Por último, el generador de niveles deberá de ser capaz de notificar al sistema de UI de la generación de una nueva habitación, para que el sistema de UI haga el procesamiento correspondiente con respecto a la nueva habitación generada a la hora de generar y enlazar las diferentes habitaciones generadas en el minimapa desde el sistema de UI.

4.2.2 Generador de habitaciones

El sistema generador de habitaciones es el encargado de generar y gestionar el contenido de las diferentes habitaciones que son elegidas aleatoriamente por el generador de niveles. Este generador se encarga de generar de manera aleatoria los diferentes enemigos que se van a encontrar en la habitación. Además, el generador de habitaciones es en el encargado de notificar al generador de niveles la posición en la que se ha generado dicha habitación, con el objetivo de que el generador de niveles sea capaz de "enlazar" las habitaciones correspondientes con dicha habitación que se acaba de generar.

4.2.3 Sistema de enemigos

El sistema de enemigos es el encargado de manejar el comportamiento de los diferentes enemigos con respecto al jugador, con relación a sus movimientos, ataques y animaciones. El sistema de enemigos es el encargado de comprobar los disparos recibidos por parte del jugador y debe de ser capaz de notificar al sistema de objetos, para generar un objeto aleatoriamente cuando el enemigo es eliminado por parte del jugador.

4.2.4 Sistema de objetos

El sistema de objetos es el encargado de manejar las colisiones de los objetos generados en el nivel debido a las eliminaciones de los diferentes enemigos con el jugador, para comprobar si es posible subir la estadística correspondiente al objeto con el que se ha detectado la colisión y en caso afirmativo, aumentar dicha estadística.

4.2.5 Sistema de jugador

El sistema de jugador es el encargado de manejar todo lo relacionado con el jugador en torno a su movimiento, su mecánica de disparo y sus estadísticas. El sistema de jugador se comunicará con el sistema de

enemigos para poder detectar los diferentes ataques realizados por los enemigos al jugador y con el sistema de objetos para detectar y comprobar si es posible aumentar la estadística correspondiente al objeto recogido. Por último, el sistema de jugador deberá de ser capaz de notificar los cambios en la vida del jugador al sistema de UI, para que este mismo se actualice y muestre dichos datos al jugador.

4.2.6 Sistema de UI

El sistema de UI es el encargado de gestionar toda la información relacionada con la interfaz del juego dentro de la interfaz de usuario con el objetivo de informar al jugador de lo que necesita saber en todo momento. El sistema de UI es capaz de comunicarse con el sistema de generador de niveles para ser consciente de las diferentes habitaciones que se están generando mediante el generador de niveles y el generador de niveles, y añadir estos tipos de habitación en formato 2D al minimapa del nivel actual para mostrar la información al jugador en todo momento donde se encuentra (teniendo en cuenta el descubrimiento por parte del jugador de las diferentes habitaciones del nivel). Además, también es capaz de comunicarse con el sistema de jugador para que este le notifique los diferentes cambios que ha sufrido la vida del jugador y así poder mantener actualizada la vida del personaje en la interfaz de usuario para dar esta información al jugador.

5. Implementación

Una vez se conoce un poco la base teórica del videojuego y el diseño que se ha realizado para su implementación, es necesario llevar a cabo todo el desarrollo de las diferentes partes propuestas del videojuego y explicar sus implementaciones.

5.1 Descripción de la solución propuesta

Aunque ya se ha explicado de forma detallada anteriormente, el sistema propuesto se puede dividir principalmente entre 3 partes diferentes:

- Sistema de generación de niveles / gamemanger: El sistema encargado relacionado con la generación de niveles y encapsulando la generación de habitaciones.
- Sistema de enemigos y jugador: Se trata de los sistemas en conjunto que se encargan de toda la lógica relacionada con el jugador y los enemigos a la hora de realizar diferentes acciones, así como, de la interacción por parte del jugador con los objetos.
- Sistema de UI: Sistema encargado de mantener una comunicación constante con los otros sistemas para mantener al jugador actualizado de forma constante con la información necesaria en cada momento.

5.2 Relaciones entre los sistemas

Las diferentes relaciones entre los sistemas se especifican en la Figura 3.

Como se puede comprobar todos los sistemas son dependientes entre sí, pues es necesario tener una comunicación constante entre ellos para poder llevar a cabo la tarea principal de ejecutar la lógica del videojuego. Estas relaciones se pueden ver entre si pues los enemigos nunca aparecerían en el juego, si el sistema de generación de niveles no existiese o no hiciera bien su trabajo y por otro lado, el jugador no sería capaz de avanzar el nivel puesto que si el sistema de generación de niveles no lograra enlazar de forma correcta las diferentes habitaciones, el jugador podría verse todo el rato en la misma habitación y no se llevaría a cabo la idea principal del juego correctamente. Además de estos dos sistemas, el más dependiente de todos es el sistema de UI, puesto que es un sistema cuyos datos y actualizaciones dependen de los otros dos mencionados anteriormente, por lo que un error cualquiera en alguno de los otros dos sistemas puede llevar a cabo que la información proporcionada por el sistema de UI al jugador no se muestre o actualice correctamente, causando que la experiencia de juego no sea correcta y el juego no se ejecute correctamente.

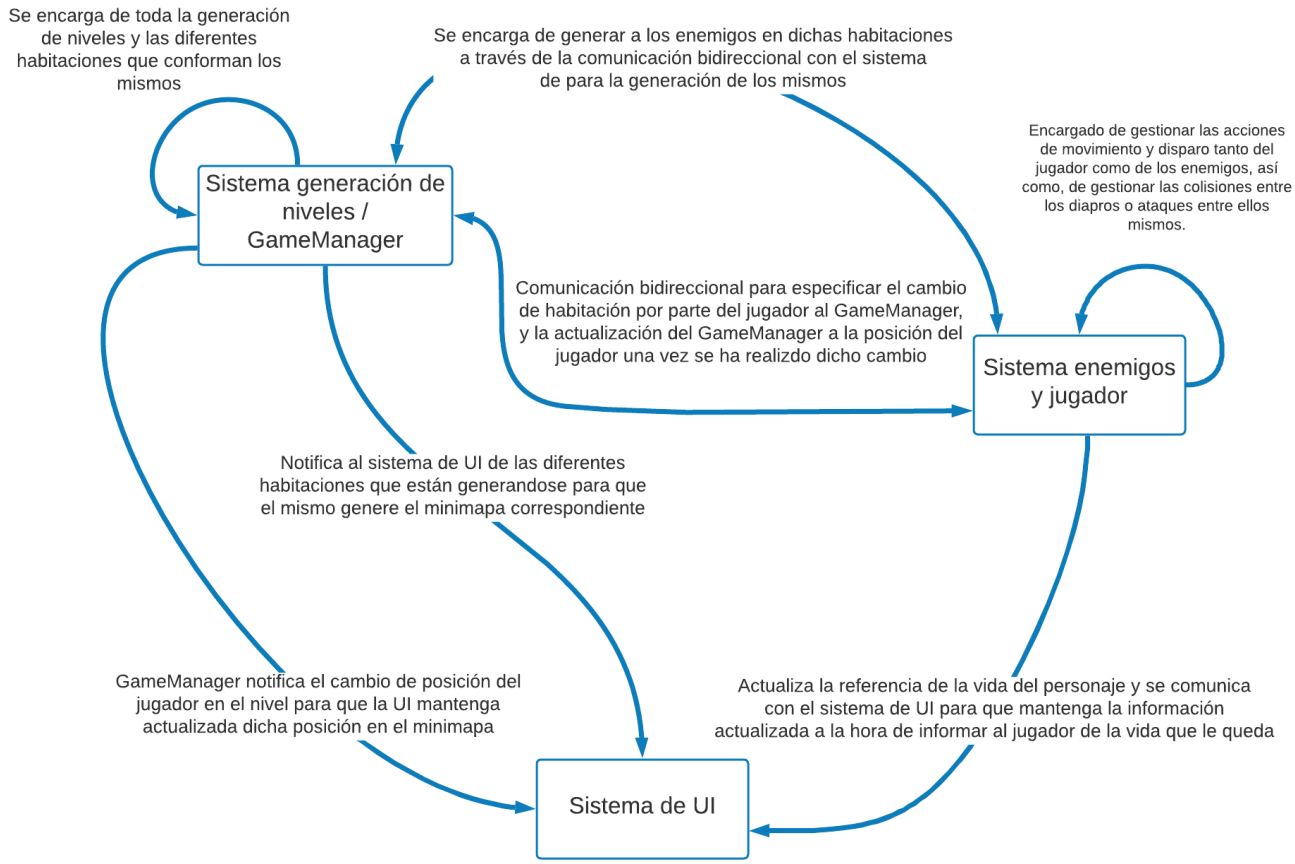

Figura 3 Relaciones entre los sitemas

5.3 Desarrollo del generador de niveles

Para el desarrollo de los niveles se han desarrollado unas habitaciones con distintos "*gameobject*" denominados como "*spawnpoints*" los cuales una vez es creada la habitación inicial, se generan nuevas habitaciones en dichas posiciones especificando el tipo de habitación que necesita para enlazar la nueva habitación con dicha habitación. Y un *gameobject* exclusivo en el centro para cerciorar al algoritmo de que esta habitación ha sido creada. Todo lo relacionado con comprobar que la habitación ha sido creada y comprobar la zona en la que se va a crear una nueva habitación, se realizan gracias a las colisiones de los "*colliders*", a través de estas colisiones se comprueba correctamente si es posible crear la nueva habitación en dicha habitación gracias a que no colisiona con el *collider* central de alguna de las habitaciones ya creadas. Esta comparación de los *collider* con respecto a las colisiones se realiza de una manera correcta gracias al uso de las etiquetas puesto que los *colliders* que nos interesan en este caso de la generación son los que tienen la etiqueta de "*SpawnPoint*"

Al instanciar el objeto de la nueva habitación, esta habitación es añadida a una estructura de datos donde se enlazan las coordenadas x y z con el *gameobject* correspondiente a la habitación que se acaba de crear. Esto se utiliza para poder enlazar las diferentes habitaciones según la posición en la que se encuentran y a la que se quiere mover.

Esto se usa en conjunto con el *GameManager*, donde se establece un número máximo de habitaciones por nivel para que cuando el número de habitaciones creadas actualmente supera el número de habitaciones máxima definida previamente, las nuevas habitaciones creadas a partir de dicho momento son creadas con el objetivo de cerrar el nivel, es decir, habitaciones con una única puerta que se enlazará con la habitación que ha generado la creación de esta nueva habitación.

Además, en esta generación de la habitación, se debe de comunicar con el sistema de UI para que realice el mismo proceso, pero en la parte de Interfaz de Usuario, es decir, escoja la miniatura correspondiente a la habitación que se acaba de generar y se coloque en el lugar correspondiente a la interfaz de usuario para que enlace de forma correcta con la habitación principal o con las habitaciones que tenga a su alrededor.

De esta forma, se puede ver la construcción de una habitación con los diferentes *colliders* mencionados anteriormente en la **Figura 4.**

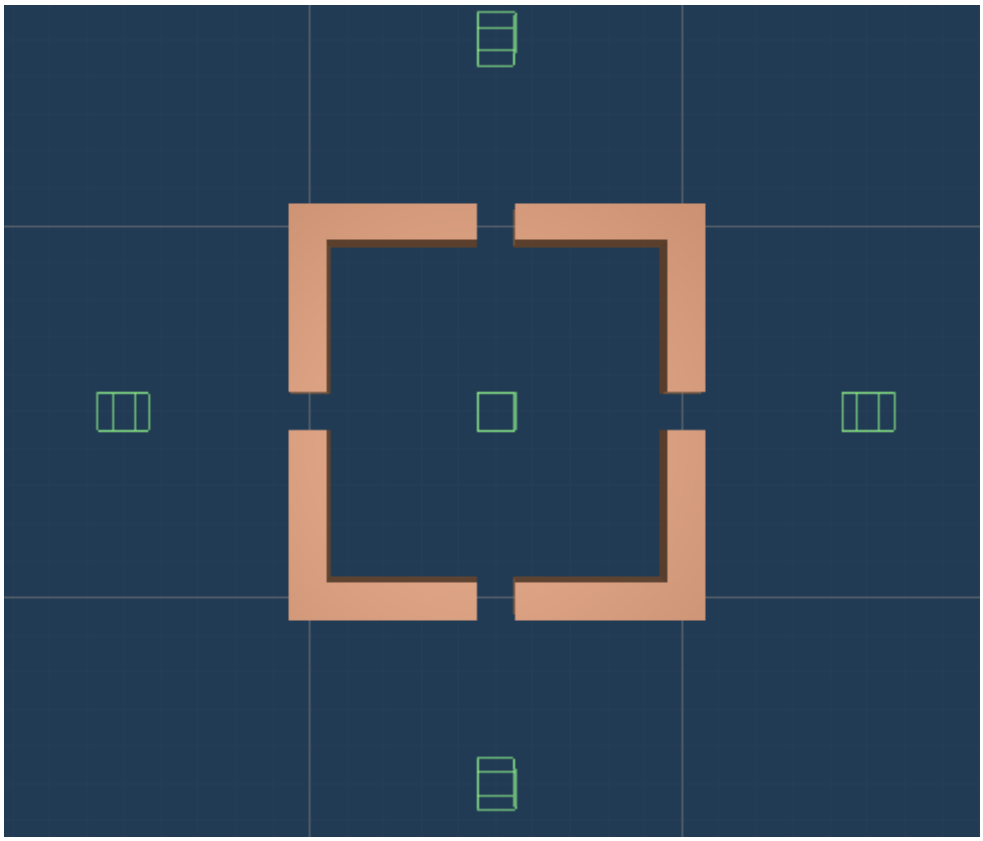

Figura 4 Ejemplo de habitación con los *colliders en color verde* como spawnpoints

Además de esto, en la misma habitación se han añadido diferentes *gameobjects* con diferentes *colliders* a los mencionados anteriormente para diferenciarlos de los mismos. Estos nuevos colliders son los encargados de comprobar la detección de la colisión del jugador con ellos mismo para cambiar de habitaciones dentro del propio nivel (véase Figura 5).

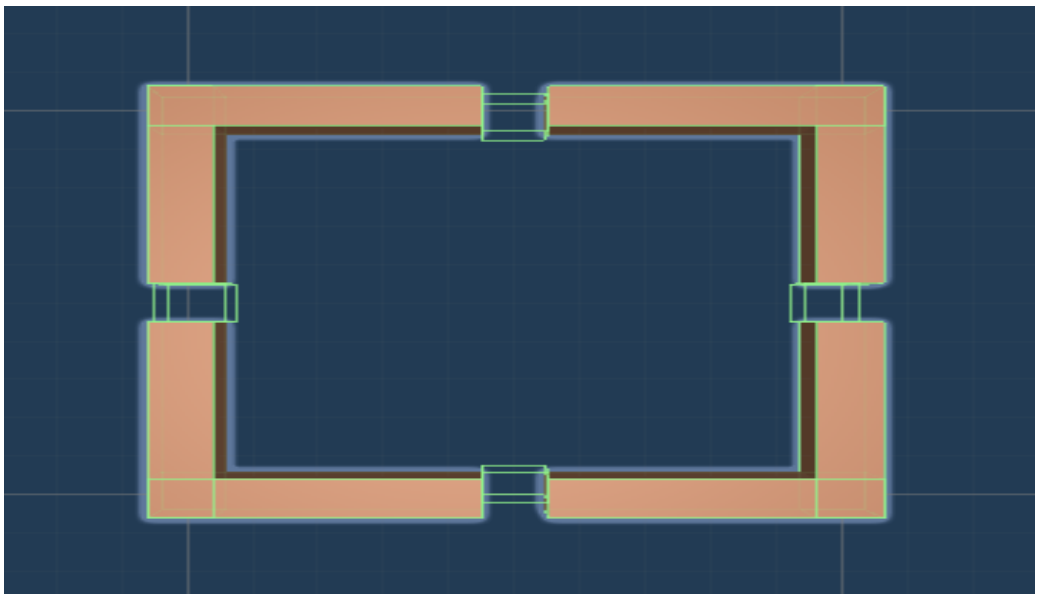

Figura 5 Colliders en habitaciones para cambiar de habitación

Estos *colliders* son los encargados de ejecutar la lógica necesaria para cambiar de habitación una vez el jugador colisiona contra los mismos. Esta lógica está basa en la comprobación de que la colisión se ha realizado con un objeto con la etiqueta de "***Player***" para así diferenciar a los demás objetos con el jugador, que es el que nos interesa en este caso. Una vez esa comprobación se realiza efectivamente, se utiliza el valor absoluto de la posición del jugador en las coordenadas x y z.

Esto se realiza para comprobar si el jugador pretende moverse a una de las habitaciones colindantes en el eje x, es decir, izquierda o derecha, o en el eje z. Por lo tanto, se utiliza el valor absoluto de ambas coordenadas para ver cuál de ambas es mayor con respecto a la posición actual de la posición, es decir, conocer si el jugador está de forma dominante en el eje z o en el eje x para conocer si se quiere mover de forma lateral o vertical entre habitaciones.

Una vez se conoce cuál de las dos coordenadas es mayor, se comprueba si dicha coordenada es mayor o inferior a la coordenada del objeto correspondiente a la habitación, para comprobar en qué dirección se quiere mover (por ejemplo, si la x es mayor indicamos que se quiere

mover hacia la habitación que se encuentra a la derecha, si es menor a la habitación que se encuentra justo a la izquierda de esta).

En el momento de cambiar de habitación, el generador de niveles se comunica con el *GameManager* para indicarle la intención de dicho cambio y que el mismo sea el encargado de cambiar la posición de todo lo necesario (jugador, cámara, etc…) y además de activar la habitación correspondiente para que los enemigos de dicha habitación empiecen a ejecutar su lógica y ataquen al enemigo.

Todo esto se realiza mediante una "pausa" del juego, para no permitir al jugador moverse hasta que no se encuentra en la nueva habitación y hasta que la pantalla no vuelve a tener en cámara al propio jugador y a la nueva habitación a la que se ha movido el mismo. Esto es debido a que, una vez el jugador cambia de habitación se activa un panel oscuro para tapar toda la pantalla, que se encuentra activo hasta que no se ha realizado todos los cálculos y los cambios necesarios para que el jugador cambie de habitación y que dicho panel desaparece una vez todo este proceso acaba, para dar la posibilidad al jugador de retomar el juego, dando así una sensación más natural con respecto el cambio de habitación al jugador, en comparación a que todo lo anterior se realizase de manera instantánea a vista del jugador.

La forma en la que se consigue esto es mediante una estructura de datos de tipo *Map* o *Dictionary* en C#. Esto se puede considerar como un array de dos dimensiones de tipo *GameObject* que nos proporciona Unity, ya que la estructura en sí se basa en dos *Dictionary*, donde el primero nos manejamos por la coordenada x de las habitaciones que se utiliza como *Key* y se inserta un nuevo *Dictionary* donde se utiliza la coordenada z como *Key* y finalmente el *GameObject* de la habitación correspondiente es asignado a dicha *Key*. Esta estructura puede verse de una forma muy similar a un array de dos dimensiones ya que se utilizan dos coordenadas de números enteros para acceder a un objeto concreto, pero se ha decidido utilizar dos estructuras de datos de tipo *Dictionary* debido al gran uso de comprobaciones de *Key* que se utiliza para saber si existe una habitación en dicha coordenada o no, y esto se realiza de una manera mucho más efectiva en estructuras de dato de tipo *Dictionary* que en estructuras como *array*. En la Figura 6 puede verse una representación en formato de array de la estructura que se ha utilizado para almacenar las diferentes habitaciones con respecto a la coordenada x y z en la que se encuentra.

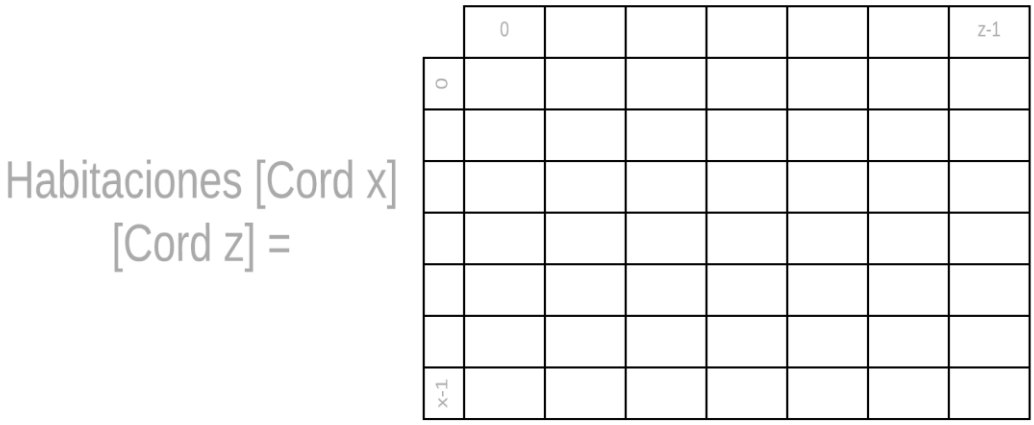

Figura 6 Ejemplo estructura de datos de habitaciones con la coordenada x como fila y la z como columnas

Con todo lo que se ha explicado anteriormente se tendría una visión completa de cómo funciona el comportamiento de la generación de habitaciones y se podría llegar a ver una plantilla completa de un ejemplo de habitación con todos los *colliders* que podría contener dicha habitación, ya que estos mismas varían según el tipo de habitación que se implemente (véase Figura 7).

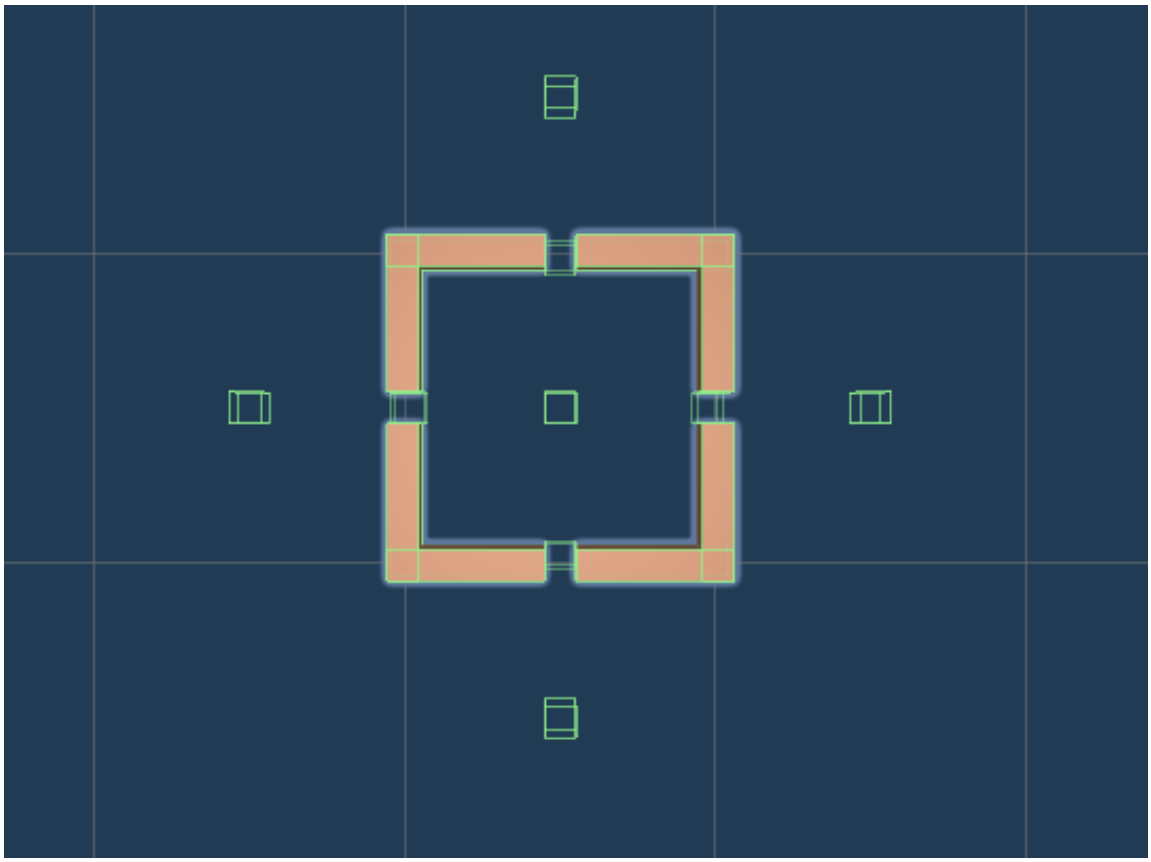

Figura 7 Ejemplo habitación inicial con todos los *colliders* necesarios para dicha habitación

Una vez esta generación de la habitación se ha completado y se ha generado e insertado en las diferentes estructuras de datos, la propia habitación se encarga de comunicarse con el sistema de enemigos, para llegar a generar de manera aleatoria los diferentes enemigos, es decir, se elige de manera aleatoria la cantidad de enemigos que se van a generar en dicha habitación y una vez se define esa cantidad se genera los diferentes enemigos de manera aleatoria y son añadidos a la referencia de la habitación que se acaba de crear.

Esto se realiza para que cuando la habitación sea destruida, se destruya de manera recursiva todas las referencias de los objetos que se encuentran en su interior (están referenciados por ellas misma) y así se destruye todo lo relacionado con dicha habitación.

Esto conlleva a que la habitación debe de contener referencias a los enemigos generados y la cantidad de estos, llevando a la situación en la que se elimine a un enemigo, la habitación debe de ser notificada de esto para eliminar dicha referencia y poder destruir al enemigo correspondiente de forma limpia y sin problemas. Esta misma generación de enemigos con respecto a la referencia de la habitación se realiza en la generación de los objetos una vez el enemigo es eliminado, es decir, el objeto que se genera debe de estar ligado mediante referencia a la habitación a la que pertenece, para que en el caso en el que el jugador decida no recoger el objeto y se pase al siguiente nivel, se llegue a limpiar la habitación sin ningún tipo de problema y se eliminen todas las referencias contenidas en la habitación de forma correcta y así no quede nada del anterior nivel en el nuevo.

Se ha explicado la forma en la que las diferentes habitaciones son generadas, y como la generación de estas habitaciones es capaz de enlazar unas con otras a través de las coordenadas de los ejes x y z para habilitar al jugador poder cambiar de habitación y así ir moviéndose por el nivel del juego. A continuación, se va a comentar la forma en la que se implementa la decisión de elegir la habitación en la que se establecerá el *boss* del nivel actual para poder pasar al siguiente nivel.

Mientras las diferentes habitaciones se van generando no solo se van añadiendo a la estructura de datos mencionada anteriormente para enlazar las habitaciones y así permitir el movimiento del jugador, si no que, además de esto también son añadidas a una lista para tener de una manera más sencilla y simple el acceso a las diferentes habitaciones generadas en el nivel.

Mientras las habitaciones se van generando y se van añadiendo a dicha lista, se establece un tiempo de espera que es el encargado de esperar hasta que las habitaciones se han terminado de generar (en este caso

se espera una cantidad fija de tiempo debido a que la generación nunca supera dicha cantidad de tiempo debido a la cantidad de habitaciones que se ha permitido por nivel). Una vez esta espera termina, se selecciona la última habitación generada como la habitación asignada al *boss* del nivel actual. Una vez esto ocurre, se establece dicha habitación como la habitación correspondiente al *boss,* para que una vez el *boss* sea derrotado en dicha habitación, aparezca la salida y se pueda generar un nuevo nivel.

Esta lógica implica eliminar los enemigos que se hubieran generado anteriormente en dicha habitación y seleccionar el enemigo como *boss*, el único enemigo que va a formar parte de esta habitación única del nivel, y de esta manera se puede comprobar de manera constante si la habitación se trata de la habitación en la que se encuentra el *boss* mediante un “flag”, y en caso afirmativo, comprobar si ha sido eliminado o no para generar el objeto correspondiente a la salida. Esta salida es un objeto bastante simple, cuya única lógica se trata de detectar la colisión por parte del jugador con dicho objeto y ser el encargado de invocar la generación de un nuevo nivel cuando se detecte dicha colisión.

La generación de un nuevo nivel cuando ya se ha creado uno previamente se basa en la generación de habitaciones ya explicada anteriormente, es decir, una vez se detecta la colisión del objeto de salida con el jugador y se invoca la generación del nuevo nivel, esto se lleva a cabo mediante la comunicación con el *GameManager*, que será el encargado de actualizar el nivel actual en el que se encuentra el jugador y de eliminar todas las referencias existentes del nivel actual.

Esta eliminación de las referencias se basa en destruir los diferentes objetos que se han ido almacenando en las diferentes estructuras de datos especificados anteriormente, es decir, se destruyen todas las habitaciones y todos los enemigos que se han generado en dichas habitaciones, así como, los diferentes objetos que se han generado debido a la eliminación de los mismos enemigos.

Una vez todas las habitaciones han sido destruidas, y con ellas, también se han destruido todas las referencias a las mismas, y sus enemigos, el *GameManager* se encarga de comunicarse con el sistema de UI para que el mismo lleve a cabo este proceso en la parte del minimapa, es decir, se eliminen todas las referencias de las diferentes habitaciones que se han generado en el nivel pero para la parte del minimapa, y de la misma manera, que todos los objetos que se hayan creado debido a la generación de estas habitaciones sean eliminados de forma instantánea.

Además, una vez se eliminan todas las referencias y objetos citados anteriormente, el sistema de UI, debe de actualizar la referencia de la posición en la que se va a encontrar el jugador en el nuevo nivel, que siempre va a corresponder a la posición 0,0,0. Una vez el sistema de UI es actualizado y todos los objetos eliminados, empieza la generación del nuevo nivel, a partir de la generación de la habitación de entrada en la

posición 0,0,0 y la actualización por parte del *GameManager* de las posiciones correspondientes al jugador y la cámara del juego, que van a ser las correspondientes a 0,0,0 al igual que la habitación principal del nivel. Por último, todo lo anterior se ha realizado, la generación de la habitación principal a través de los diferentes objetos con la etiqueta de "***SpawnPoint***" es la encargada de generar las diferentes habitaciones como se ha explicado anteriormente.

5.4 Desarrollo del sistema de enemigos

Para el desarrollo del sistema de enemigos, se han desarrollados cuatro tipos diferentes de enemigos comunes:

- Slime
- Tanque
- Ranger
- Ranger2

Estos enemigos, aunque comparten un comportamiento de manera común con respecto a ciertas propiedades, cada uno tiene un comportamiento específico diferente que se va a especificar a continuación.

5.4.1 Slime

El enemigo ***Slime*** se puede definir como un enemigo pequeño y algo veloz que se encarga de atacar al jugador cuerpo a cuerpo, sacrificándose a la hora de atacar a cambio de realizar una cantidad de daño considerable al jugador, el diseño de este enemigo ha sido obtenido del paquete de ***Assets*** otorgado por Unity [5] (véase Figura 8).

Figura 8 Modelo enemigo Slime

El comportamiento de este enemigo denominado ***Slime*** se lleva a cabo gracias al uso de los componentes de **NavMeshAgent** y **Collider**. A través, del componente de **NavMeshAgent** se otorga al enemigo una inteligencia artificial básica, la cual se usa para seguir al jugador, es decir, se establece la posición de destino a la posición actual del jugador en cada momento, para dar la sensación de seguimiento constante por parte del jugador. Además, en cada actualización de fotograma, se actualiza la rotación del enemigo, para que en todo momento este mismo este mirando todo el rato hacia el jugador y generar una sensación más realista de seguimiento.

Una vez explicado el seguimiento, quedaría explicar la parte del ataque, el cual se basa en detectar la colisión del enemigo con el jugador a través del uso de etiquetas como ya se ha mencionado anteriormente, con los ***Colliders*** de ambos objetos.

Una vez se detecta dicha colisión, la lógica de este enemigo se comunica con el sistema del jugador para contabilizar el daño que le realiza en la vida y es destruido debido a lo que ya se ha explicado anteriormente, es decir, una vez el enemigo hace daño, se sacrifica.

El modelo y la estructura completa del enemigo con los componentes de ***Collider*** y ***NavMeshAgent*** puede verse en la Figura 9 .

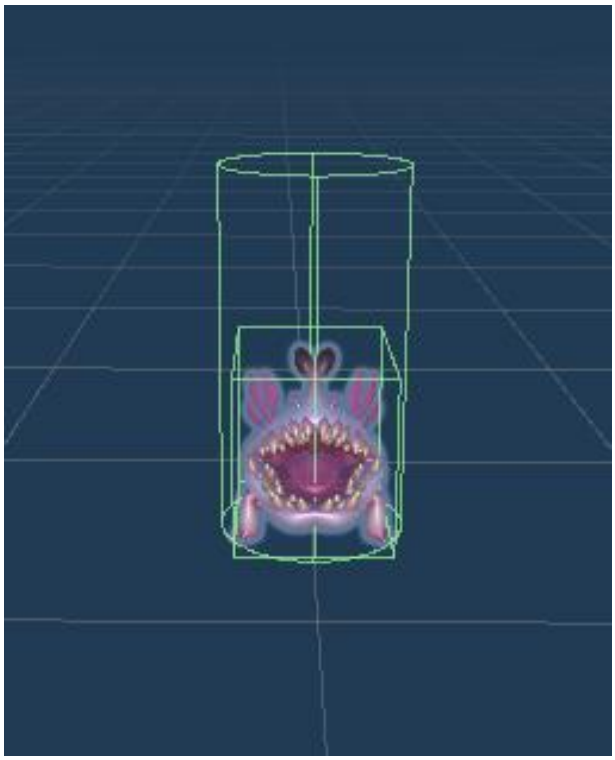

Figura 9 Estructura y modelo completo del enemigo Slime

5.4.2 Tanque

El enemigo ***Tanque*** se puede definir como un enemigo caracterizado por su defensa y lentitud, es decir, a diferencia del ***Slime***, el ***Tanque*** se basa en un enemigo muy robusto pero muy lento en consecuencia a esto por lo que realiza una gran cantidad de daño cuerpo a cuerpo al jugador si le llega a golpear, el diseño de este enemigo ha sido obtenido del paquete de ***Assets*** otorgado por Unity [5] (véase Figura 10).

Figura 10 Modelo enemigo Tanque

El comportamiento de este enemigo es muy similar al del enemigo ***Slime***, que se lleva a cabo gracias al uso de los componentes de **NavMeshAgent** y **Collider**. A través, del componente de **NavMeshAgent** se otorga al enemigo una inteligencia artificial básica, la cual se usa para seguir al jugador, es decir, se establece la posición de destino a la posición actual del jugador en cada momento, para dar la sensación de seguimiento constante por parte del jugador, pero con una excepción, que se realiza cuando la distancia entre el enemigo y el jugador es menor que un valor predefinido, dando como resultado que cuando esta distancia sea menor que el valor predefinido, se deje de actualizar la referencia del destino del enemigos con la posición del jugador y se establezca el estado del componente **Animator** para que el enemigo realice la animación correspondiente al ataque en dicho momento.

Además, en cada actualización de fotograma, se actualiza la rotación del enemigo, para que en todo momento este mismo este mirando todo el rato hacia el jugador y generar una sensación más realista de seguimiento.

Una vez explicado el seguimiento, quedaría explicar la parte del ataque, el cual se basa en detectar la colisión del enemigo con el jugador a través del uso de etiquetas como ya se ha mencionado anteriormente, con los ***Colliders*** de ambos objetos.

Una vez se detecta dicha colisión, la lógica de este enemigo se comunica con el sistema del jugador para contabilizar el daño que le realiza en la vida y en este caso a diferencia del enemigo **Slime,** el enemigo no es destruido si no ha sufrido bastante daño anteriormente, es decir, en el momento de detecta la colisión el enemigo sufre un daño predefinido y se comprueba si ese daño es suficiente como para matarlo.

El modelo y la estructura completa del enemigo con los componentes de ***Collider*** y ***NavMeshAgent*** puede verse en la Figura 11.

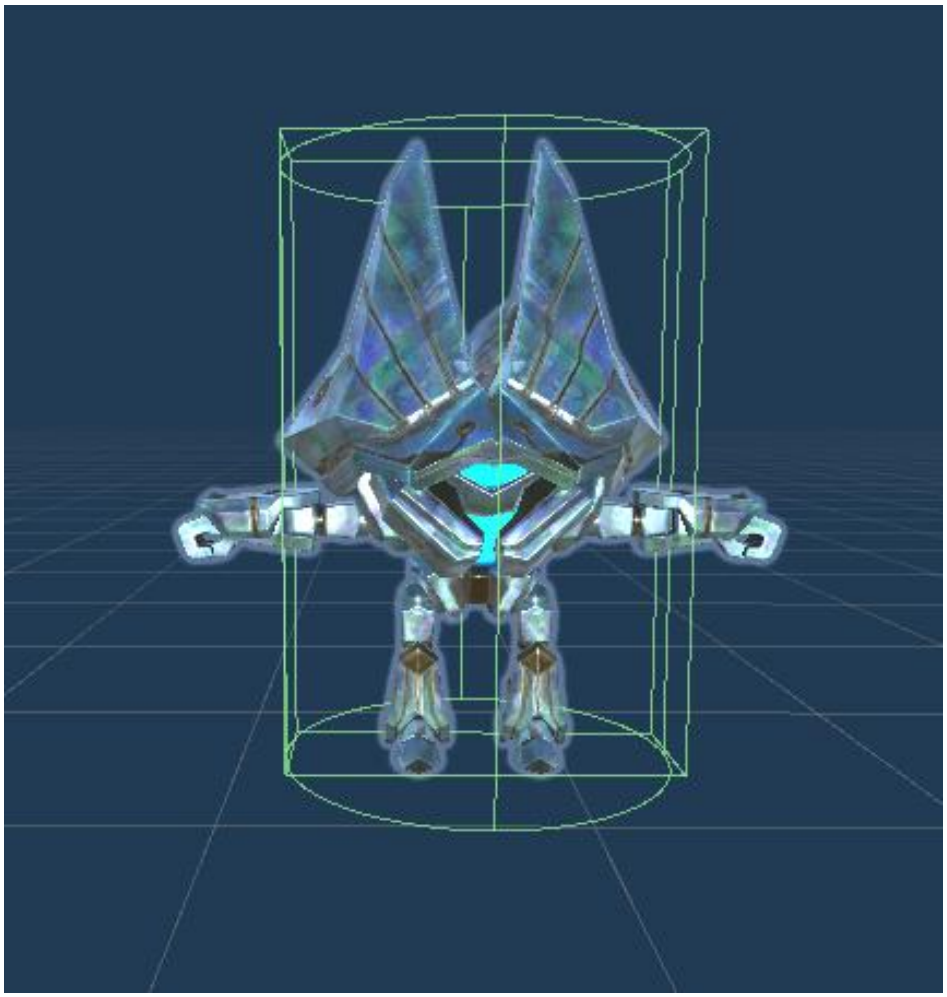

Figura 11 Estructura y modelo completo del enemigo Tanque

5.4.3 Ranger

El enemigo ***Ranger*** se puede definir como un enemigo caracterizado por disparar proyectiles directamente al jugador para hacerle daño, el diseño de este enemigo ha sido obtenido del paquete de ***Assets*** otorgado por Unity [5] (véase Figura 12).

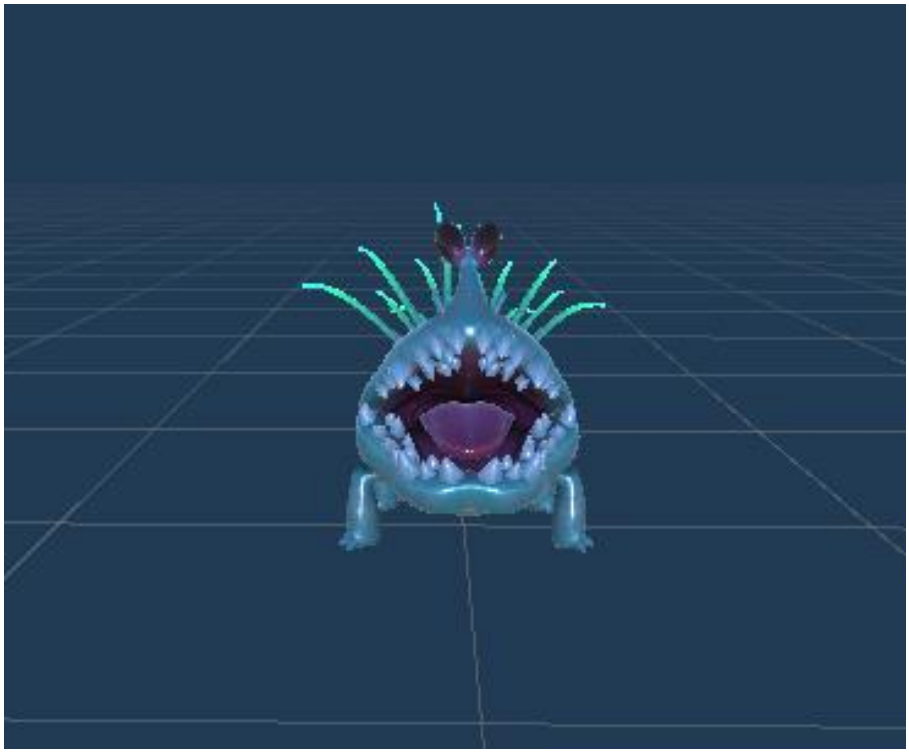

Figura 12 Modelo enemigo Ranger

El comportamiento de este enemigo denominado ***Ranger*** se lleva a cabo gracias al uso de los componentes de **Collider**. El comportamiento de este enemigo se basa en generar proyectiles que son lanzados en dirección a la posición del jugador en el momento de disparo.

Para llevar a cabo este comportamiento, se comprueba en cada actualización del fotograma si la distancia entre el jugador y el enemigo es menor que un valor predefinido, tras esta comparación, si dicha distancia es menor, se ejecuta una animación específica y se realiza el disparo si es posible.

Para comprobar si es posible lanzar un proyectil hacia el jugador, se realiza con un límite de tiempo entre disparo y disparo, calculando la diferencia de este límite de tiempo para saber si se ha superado o no, con la diferencia entre el tiempo en el que se ha realizado el último disparo y el tiempo actual que lleva ejecutándose el juego. Además, cada vez que se realiza un disparo, se modifica la rotación del enemigo para que en el momento del disparo apunte hacia la posición del jugador en dicho momento.

Para generar este disparo, se ha implementado y diseñado un objeto (“***prefab***”) que se utiliza como plantilla para generar los diferentes proyectiles que van a ser disparados hacia el jugador. El modelo de este proyectil puede ver en la Figura 13.

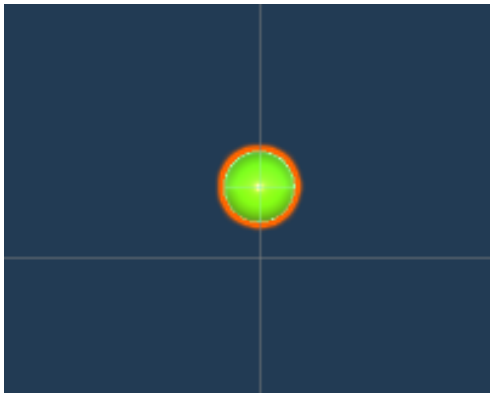

Figura 13 Modelo y estructura del proyectil usado por Ranger

Este objeto cuenta con un ***Collider*** el cual es usado para detectar las colisiones con el jugador y cuenta con una etiqueta especial denominada “**EnemyBullet**” para que una vez se detecte la colisión por parte del jugador y del proyectil, se compruebe correctamente con que se está colisionando.

El modelo y la estructura completa del enemigo con el componente de **Collider** puede verse en la Figura 14.

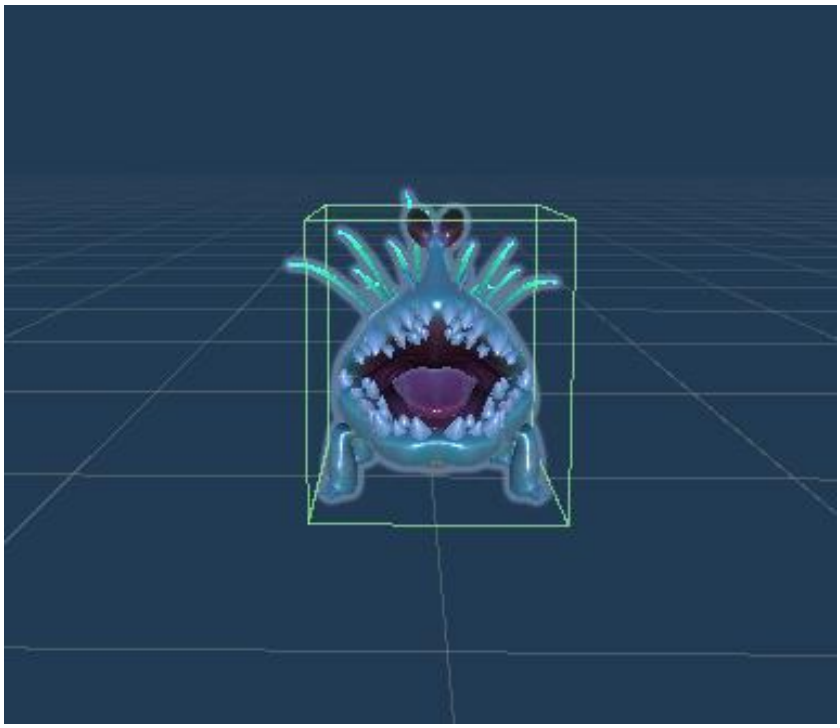

Figura 14 Estructura y modelo completo del enemigo Ranger

5.4.4 Ranger2

El enemigo ***Ranger2*** se puede definir como un enemigo caracterizado por disparar proyectiles directamente al jugador para hacerle daño, pero a diferencia con el enemigo denominado como **Ranger**, este enemigo no lanza simples proyectiles para dañar al jugador, en cambio, lanza múltiples proyectiles en forma de onda expansiva cuando detecta al propio jugador a una cierta distancia, el diseño de este enemigo ha sido obtenido del paquete de ***Assets*** otorgado por Unity [5] (véase Figura 15).

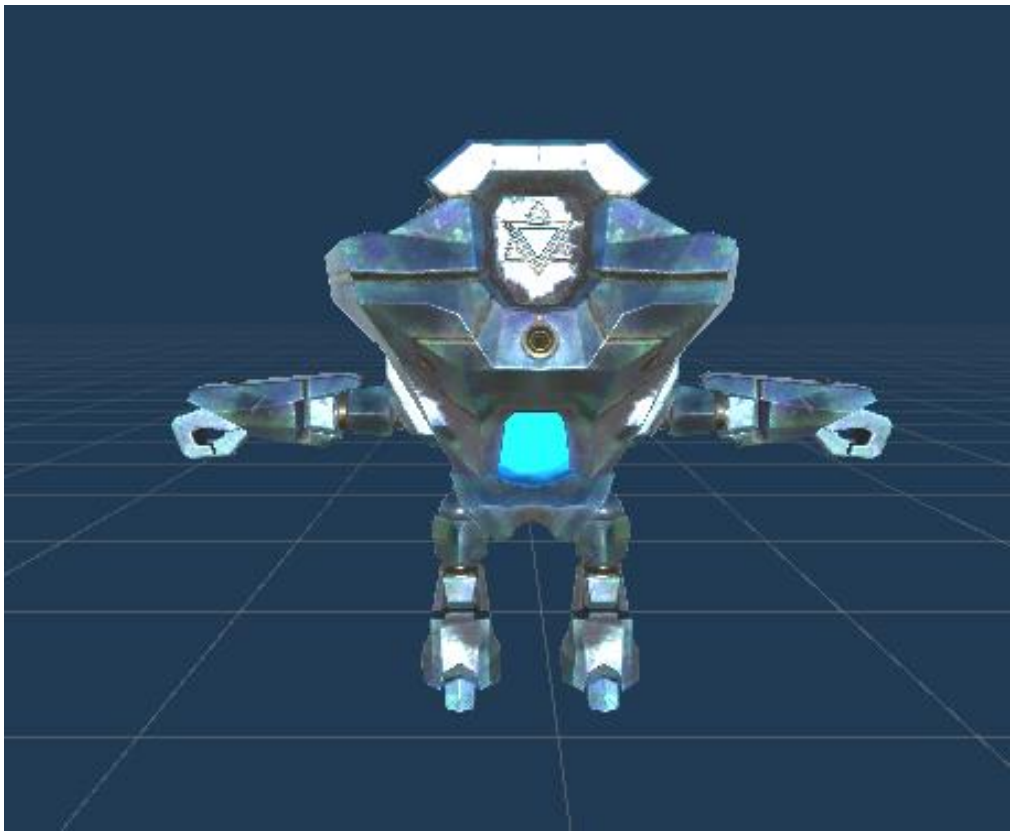

Figura 15 Modelo enemigo Ranger2

El comportamiento de este enemigo denominado ***Ranger2*** se lleva a cabo gracias al uso de los componentes de **Collider**. El comportamiento de este enemigo se basa en generar proyectiles que son lanzados en dirección a la posición del jugador en el momento de disparo.

Para llevar a cabo este comportamiento, se comprueba en cada actualización del fotograma si la distancia entre el jugador y el enemigo es menor que un valor predefinido, tras esta comparación, si dicha distancia es menor se realiza el disparo si es posible.

Para comprobar si es posible lanzar un proyectil hacia el jugador, se realiza con un límite de tiempo entre disparo y disparo, calculando la diferencia de este límite de tiempo para saber si se ha superado o no,

con la diferencia entre el tiempo en el que se ha realizado el último disparo y el tiempo actual que lleva ejecutándose el juego. Además, cada vez que se realiza un disparo, se modifica la rotación del enemigo para que en el momento del disparo apunte hacia la posición del jugador en dicho momento.

Para generar este disparo, se ha implementado y diseñado un objeto ("***prefab***") que se utiliza como plantilla para generar los diferentes proyectiles que van a ser disparados hacia el jugador. El modelo de este proyectil puede ver en la Figura 16.

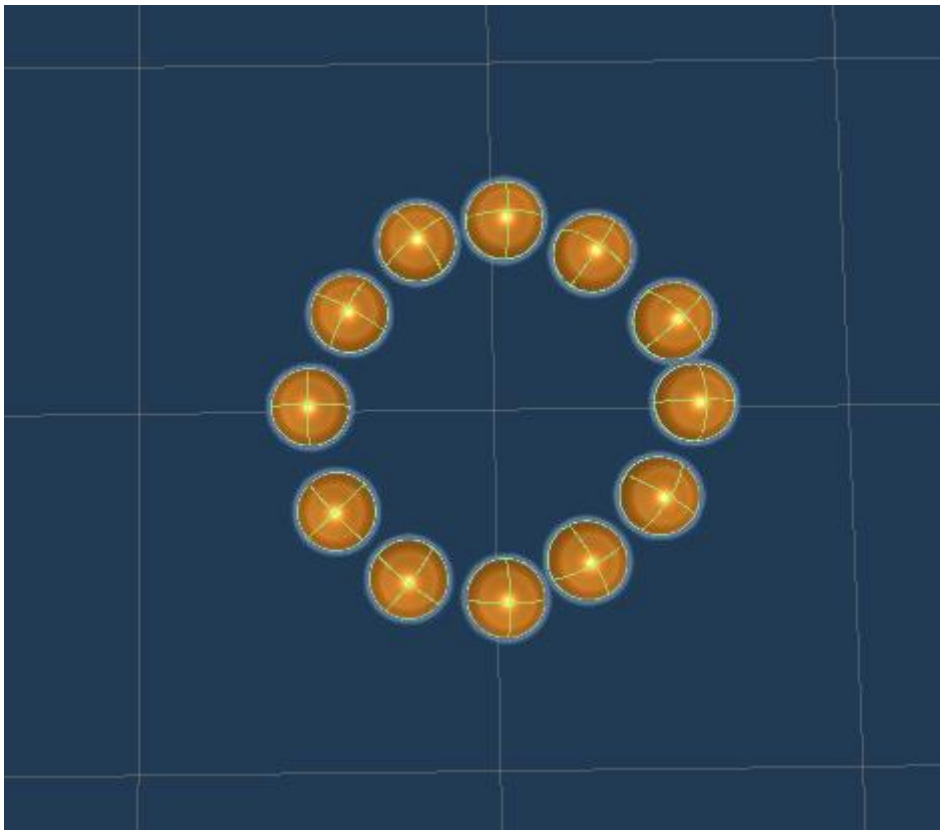

Figura 16 Modelo y estructura de los proyectiles usados por Ranger2

Este objeto cuenta con un ***Collider*** el cual es usado para detectar las colisiones con el jugador y cuenta con una etiqueta especial denominada "**EnemyBullet**" para que una vez se detecte la colisión por parte del jugador y del proyectil, se compruebe correctamente con que se está colisionando.

El modelo y la estructura completa del enemigo con el componente de **Collider** puede verse en la Figura 17.

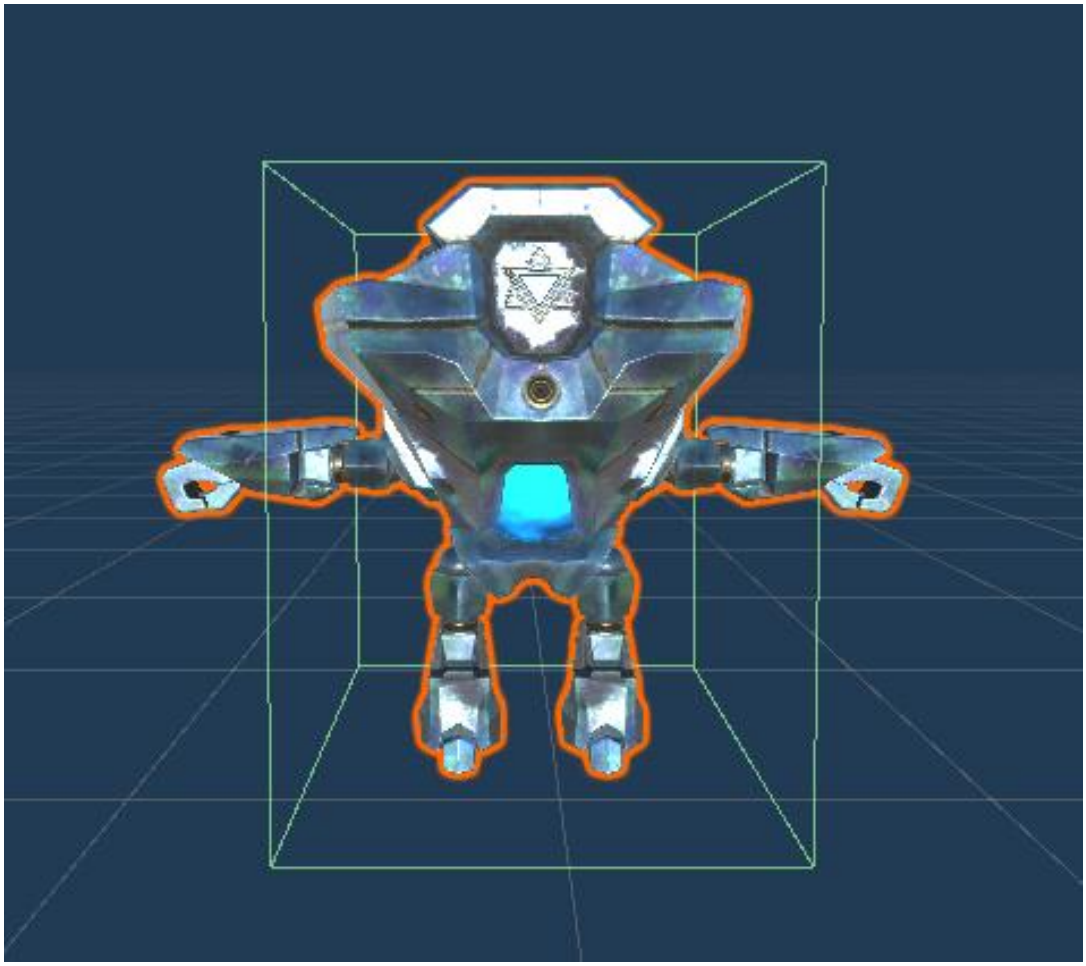

Figura 17 Estructura y modelo completo del enemigo Ranger2

5.4.5 Thanegor (Boss)

El enemigo ***Thanegor*** es el único boss implementado en el juego, y se puede definir como un enemigo grande y veloz que se encarga de atacar al jugador cuerpo a cuerpo, realizando diferentes tipos de ataque de una manera aleatoria y causando un daño considerable al jugador, para poder pasar del nivel actual primero es necesario eliminar a **Thanegor** en cada nivel (solo se cuenta con **Thanegor** como *boss* debido a que no se han implementado más enemigo de tipo *boss*), el diseño de este enemigo ha sido obtenido desde **Mixamo** de **Adobe** [2] (véase Figura 18).

Figura 18 Modelo de Thanegor

El comportamiento de este enemigo denominado ***Thanegor*** se lleva a cabo gracias al uso del componente **Collider**. A través, de este componente se detectan las diferentes colisiones por parte de los ataques de **Thanegor** al jugador.

El comportamiento de **Thanegor**, se basa en perseguir al jugador en todo momento realizando diferentes tipos de ataque a través de animaciones que se eligen aleatoriamente, es decir, a través del uso de **Corrutinas** en **Unity**, se especifica las diferentes animaciones que se van a utilizar y por tanto el ataque que va a realizar **Thanegor** en dicho momento. Estos ataques son seleccionados de manera aleatoria, pero con una excepción, la cual se basa en aumentar un contador de los ataques consecutivos de cada tipo que se realiza, para así dar más sensación de realismo, y que por aleatoriedad no se de el caso de que llegue a ejecutarse durante todo momento el mismo ataque.

Una vez se detecta la colisión por parte de **Thanegor** con el jugador, la lógica implementada para este enemigo es capaz de comunicarse con el sistema de jugador para indicarle la cantidad de daño que el jugador va a recibir por dicho ataque.

El modelo y la estructura completa del enemigo con los componentes de ***Collider*** y ***NavMeshAgent*** puede verse en la Figura 19.

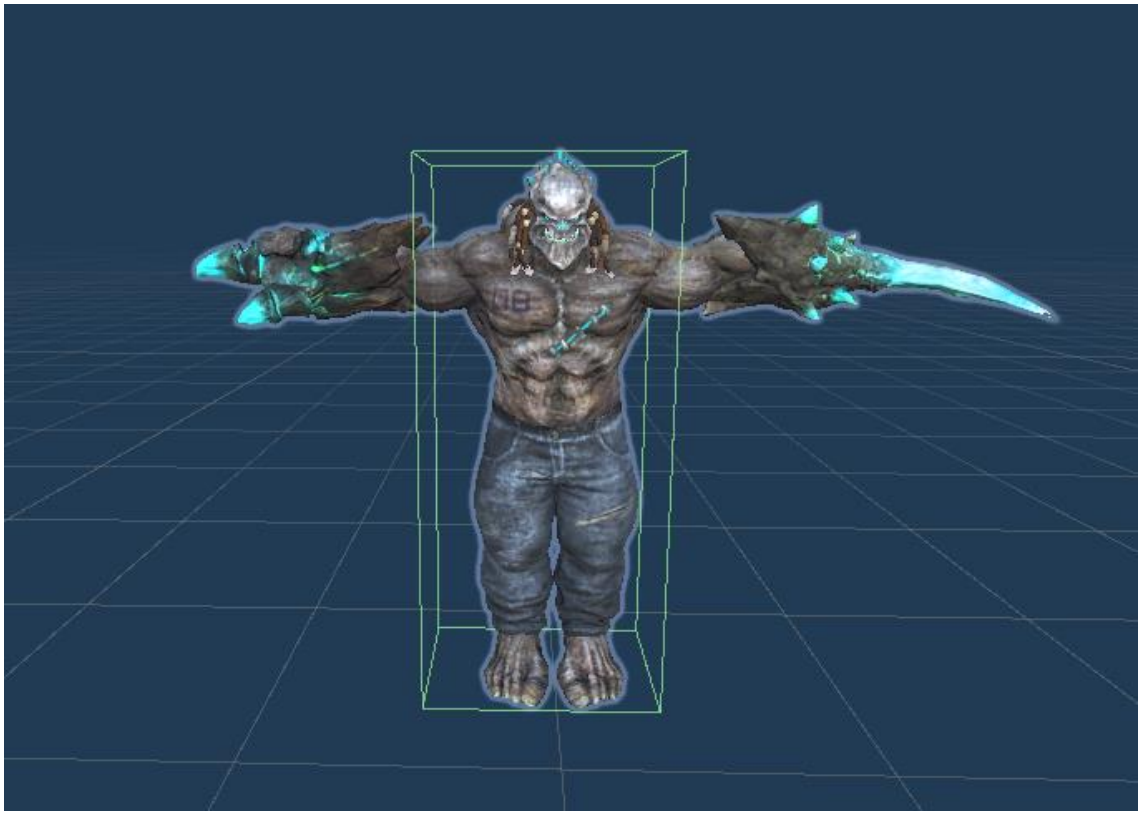

Figura 19 Estructura y modelo de Thanegor

5.4.6 Objetos

Para el desarrollo del sistema de objetos, se han desarrollados tres tipos diferentes de objetos comunes:

- Modificador de tiro
- Manzana
- Velocidad

Estos objetos se han implementado con la idea general de colisionar con el jugador de manera física, de manera que si el jugador no puede aumentar sus estadísticas teniendo en cuenta el objeto con el que está colisionando (por ejemplo, tiene la vida máxima y está colisionando con un objeto tipo **Manzana**), sea capaz de empujar al mismo objeto sobre la superficie del nivel y así dotar de aplicación al objeto sobre el jugador solo cuando sea posible aumentar las estadísticas de este.

5.4.7 Manzana

La implementación del objeto denominado como **Manzana** (anteriormente denominado como Corazón) se basa en el objetivo de

aumentar la salud actual en el momento de detectar la colisión del jugador con el objeto si este aumento es posible. El diseño de este objeto ha sido obtenido de un paquete de **Assets** gratuito especificado en la **Asset Store** de **Unity** [1] (véase Figura 20).

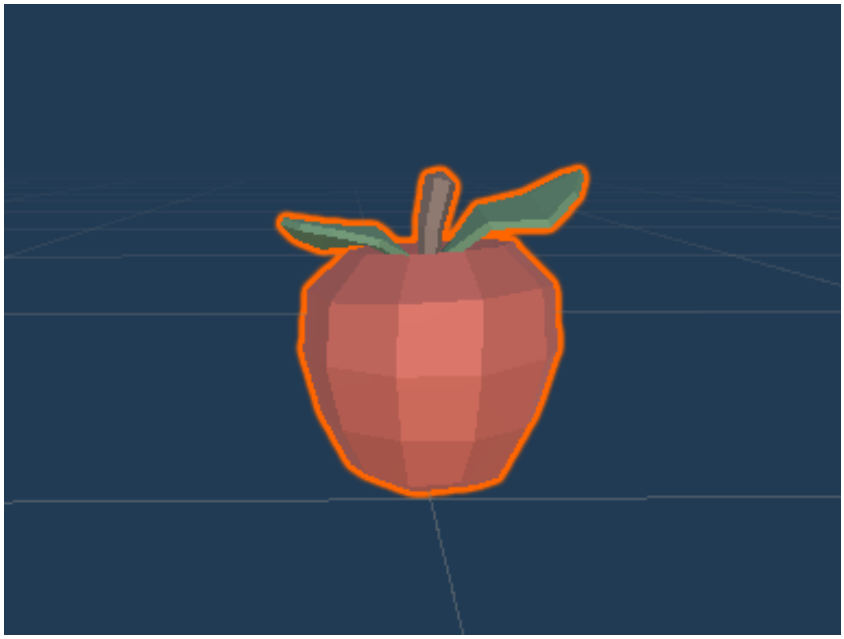

Figura 20 Modelo objeto Manzana

Debido a que la implementación de estos objetos se basa en detectar la colisión de estos con el propio jugador, el objeto cuenta con un componente de tipo **Collider**, a través del cual se compara si el objeto con el que se detecta la colisión se trata del jugador a través del uso de etiquetas. Este objeto cuenta con una cantidad fija con respecto a la salud que se le va a otorgar y una vez se detecta la colisión debe de ser capaz de comunicarse con el sistema de jugador para comprobar si es posible añadirle salud, en caso afirmativo se le añade y en caso contrario no tiene ningún tipo de efecto sobre el jugador.

El modelo y la estructura completa del objeto con el componente de **Collider** puede verse en la Figura 21.

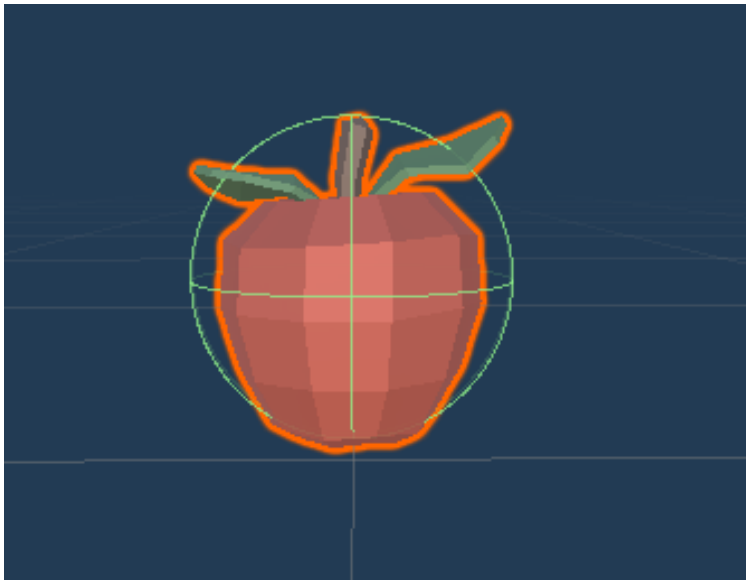

Figura 21 Estructura y modelo objeto Manzana

5.4.8 Modificador de tiro

La implementación del objeto denominado como **Modificador de tiro** se basa en el objetivo de modificar el tipo de disparo del jugador, es decir, este objeto debe de ser capaz de comunicarse con el sistema de jugador para indicarle el cambio de proyectil que va a utilizar una vez se ha recogido el objeto. El diseño de este objeto ha sido obtenido de un paquete de **Assets** gratuito especificado en la **Asset Store** de **Unity** [1] (véase Figura 22).

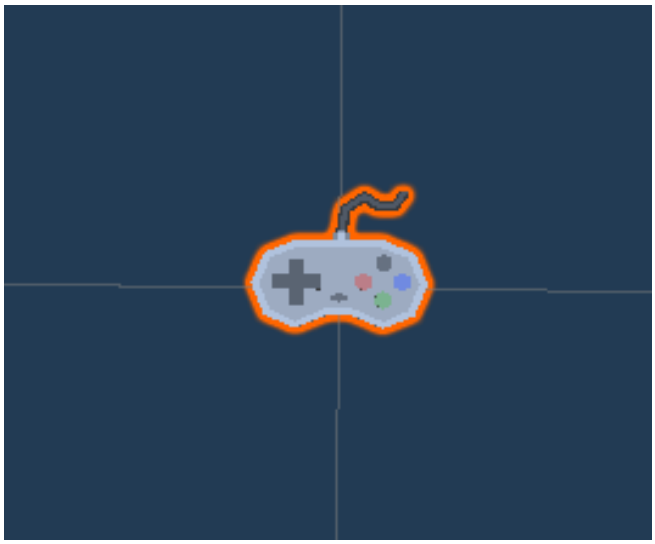

Figura 22 Modelo objeto modificador de tiro

Debido a que la implementación de estos objetos se basa en detectar la colisión de estos con el propio jugador, el objeto cuenta con un componente de tipo **Collider**, a través del cual se compara si el objeto con el que se detecta la colisión se trata del jugador a través del uso de

etiquetas. Este objeto cuenta con un tiro especifico al que cambiar el estilo de disparo por parte del jugador (esto es debido a que solo se ha especificado un tipo de tiro especial, en caso de haber modificado más, se podría haber implementado una elección aleatoria del tipo de disparo al que modificar por parte del jugador) y una vez se detecta la colisión debe de ser capaz de comunicarse con el sistema de jugador para cambiar el tipo de disparo actual que tiene el jugador.

Para implementar este tipo de disparo especial, se ha diseñado e implementado un objeto especifico con diferentes balas que se utiliza como "***prefab***" en los tipos de disparos implementados y aportados al jugador. Este tipo de disparo especial se basa en cuatro proyectiles, los cuales cada uno se desplaza en los ejes x y z, pero en diferentes direcciones formando como una especie de cruz. El modelo de estos cuatro proyectiles en conjunto puede verse en la Figura 23.

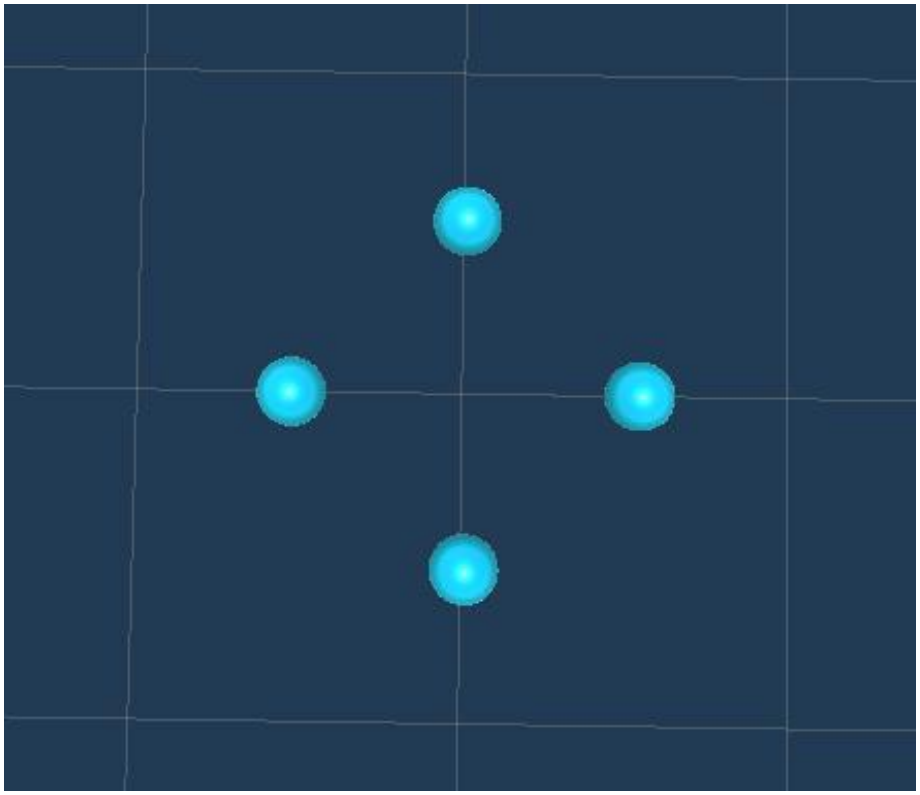

Figura 23 Modelo del tiro especial implementado para el modificador de tiro

El modelo y la estructura completa del objeto con el componente de **Collider** puede verse en la Figura 24.

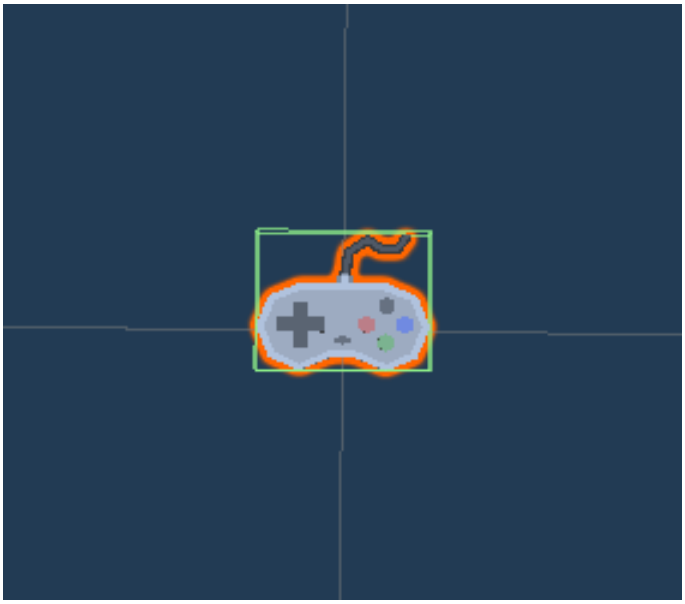

Figura 24 Estructura y modelo objeto Modificador de tiro

5.4.9 Velocidad

La implementación del objeto denominado como **Velocidad** (anteriormente denominado como Botas) se basa en el objetivo de aumentar la velocidad actual del jugador en el momento de detectar la colisión por parte del jugador con el objeto. El diseño de este objeto ha sido obtenido de un paquete de **Assets** gratuito especificado en la **Asset Store** de **Unity** [1] (véase Figura 25).

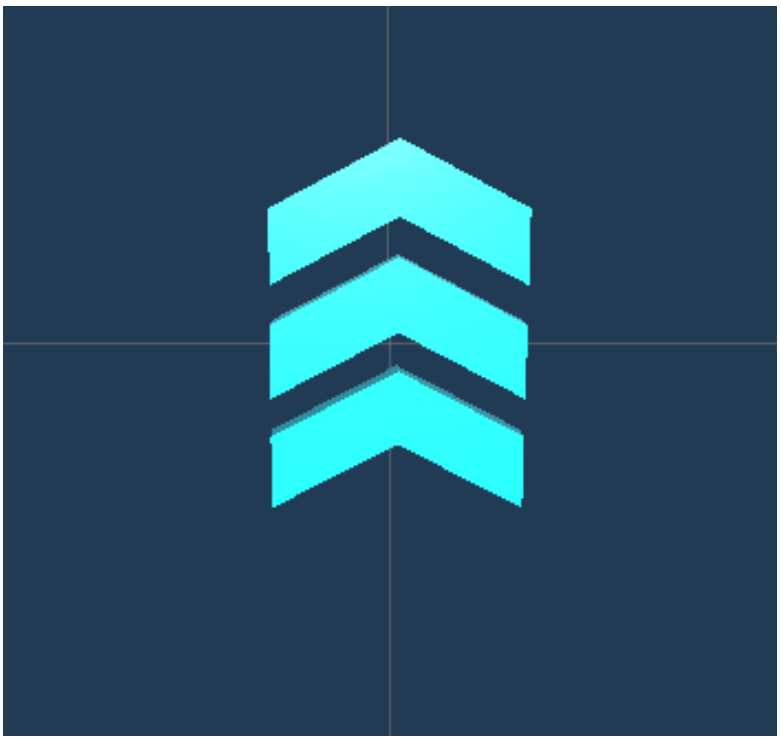

Figura 25 Modelo objeto Velocidad

Debido a que la implementación de estos objetos se basa en detectar la colisión de estos con el propio jugador, el objeto cuenta con un componente de tipo **Collider**, a través del cual se compara si el objeto con el que se detecta la colisión se trata del jugador a través del uso de etiquetas. Este objeto cuenta con una cantidad fija con respecto a la velocidad que se le va a otorgar y una vez se detecta la colisión debe de ser capaz de comunicarse con el sistema de jugador para otorgarle la cantidad predefinida de velocidad al jugador.

El cambio de nombre de este objeto ha sido debido a la dificultad de encontrar unos **Assets** con el modelo y textura de unas botas, y se ha decidido usar este modelo debido al parecido que se le puede dar con la relación al aumento de velocidad del personaje.

El modelo y la estructura completa del objeto con el componente de **Collider** puede verse en la Figura 26.

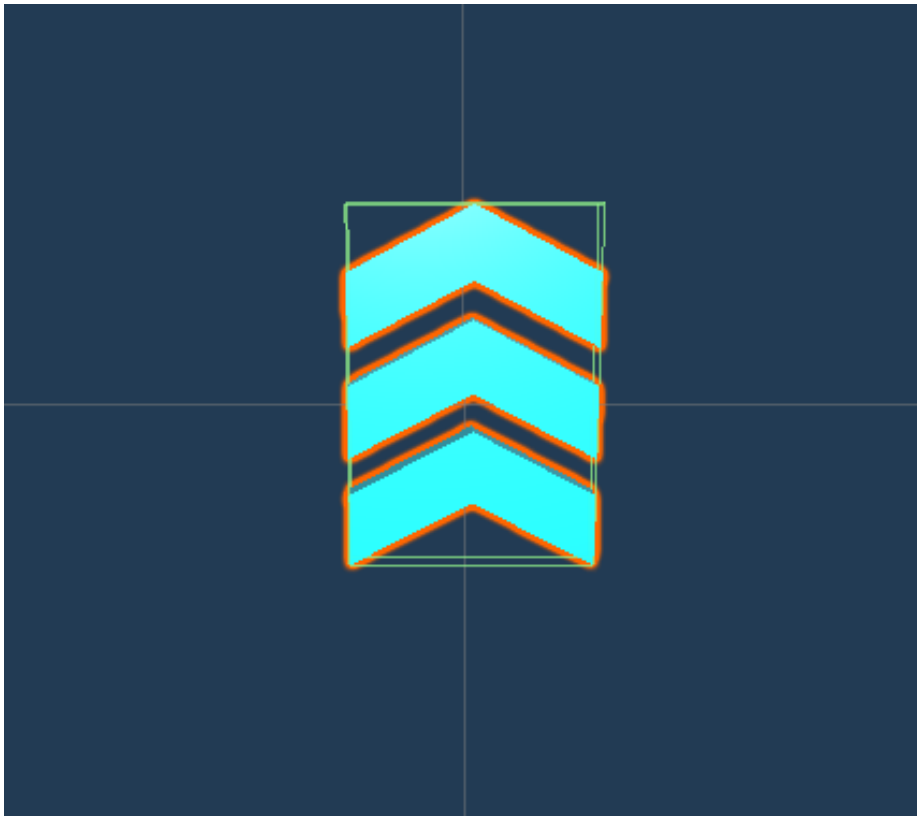

Figura 26 Estructura y modelo del objeto Velocidad

5.5 Desarrollo del sistema de jugador

Para la implementación del jugador se ha desarrollado los controles a través de la actualización de los fotogramas junto con las físicas que nos otorga Unity ("**FixedUpdate**"). Para calcular este movimiento se obtienen los valores de entrada del movimiento de los ejes tanto vertical como horizontal y son multiplicados por la velocidad actual del jugador junto con el tiempo actual que ha transcurrido en el juego desde la última

actualización de fotograma, para llegar a tener un movimiento más realista y suave. El diseño del jugador ha sido obtenido de un paquete de **Assets** gratuito especificado en la **Asset Store** de **Unity** [1] (véase Figura 27).

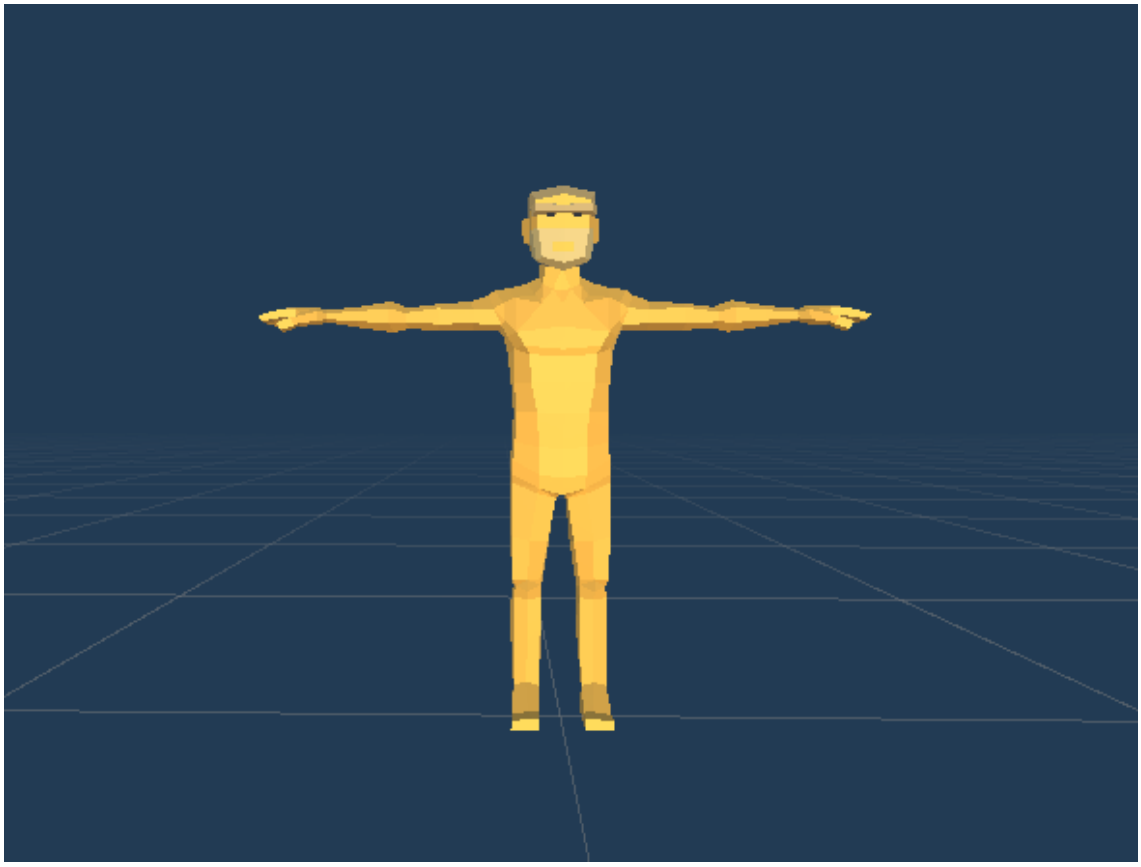

Figura 27 Modelo del personaje del jugador

Además de esto, para cada tipo de movimiento, es decir, para el movimiento lateral y el movimiento vertical se han desarrollado diferentes animaciones, las cuales son especificadas según el valor de entrada de los inputs mencionados anteriormente para encontrar en todo momento el estado correspondiente a la animación que tiene ejecutar en dicho momento. Estas animaciones se han desarrollado con **UMotionPro** [3] para el cual se ha obtenido una licencia personal, y a través de este **Asset** se ha podido generar los diferentes "huesos" para generar las animaciones desde cero a partir de esta herramienta. La estructura de los diferentes huesos que se han establecido para el desarrollo de las diferentes animaciones se puede ver en la Figura 28.

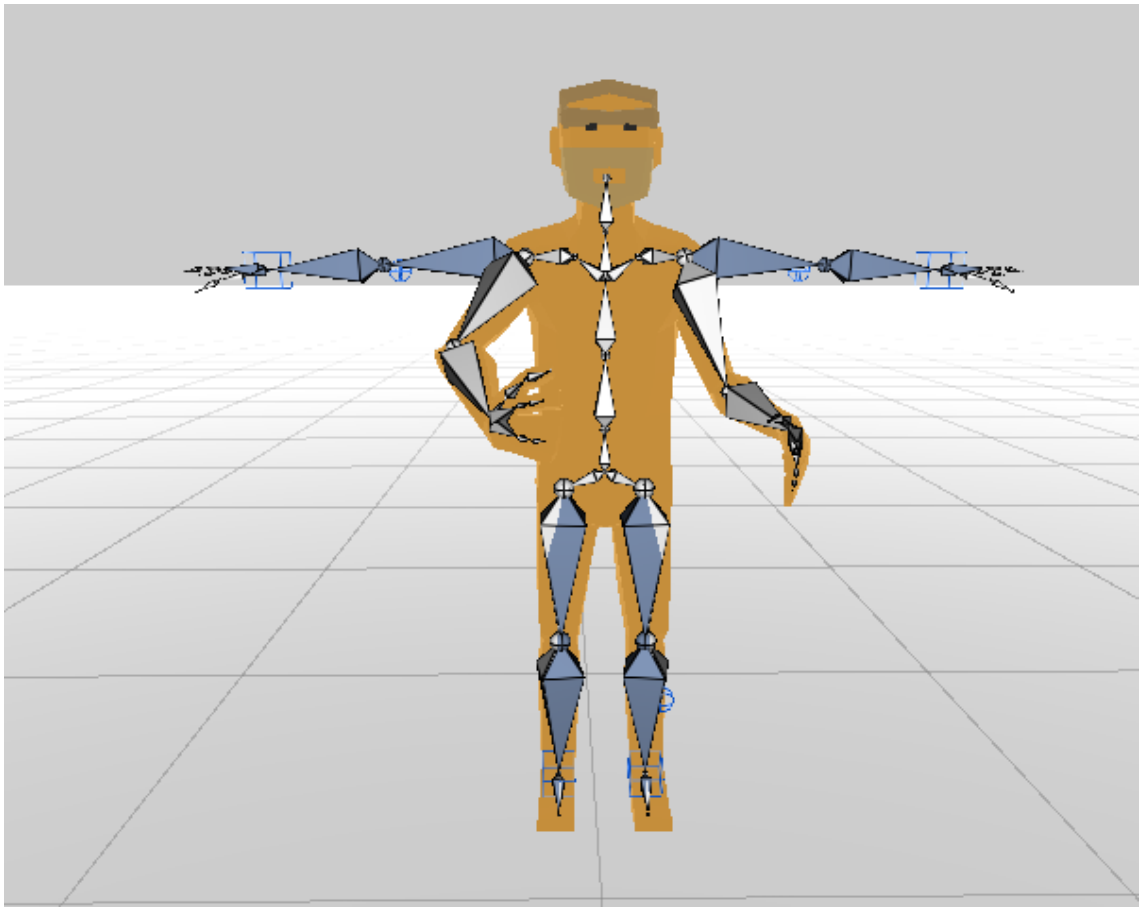

Figura 28 Estructura de los huesos del jugador generados en UMotionPro

Por otro lado, para dotar al jugador de la posibilidad de disparar proyectiles, al igual que con los enemigos, se le ha implementado una especie de límite de tiempo ("*cooldown*") para que no dispare de forma continua y sin parar y añadir un poco de dificultad al jugador. Teniendo en cuenta a esto, se comprueba si el jugador puede llegar a disparar y en caso afirmativo no solo dispara el proyectil, sino que se rota en la dirección a la que dispara.

Esto se basa en el uso de las diferentes teclas con las flechas de dirección del teclado que son usadas para disparar, por lo tanto, si se quiere disparar un proyectil hacia la derecha se utilizaría la tecla con la flecha hacia la derecha y el jugador dispararía el proyectil hacia esa dirección además de rotar el personaje para que apunte a dicha dirección al mismo tiempo, y una vez realizado el disparo se guarda el valor del tiempo en el cual se ha realizado dicho disparo para tenerlo en cuenta a la hora de comprobar si es posible realizar un disparo o no.

Anteriormente ya se ha especificado cual es el tipo de objeto que se utiliza para el tipo de disparo especial, pero a continuación se muestra el modelo básico de disparo que se le otorga al jugador desde el principio del juego (véase Figura 29).

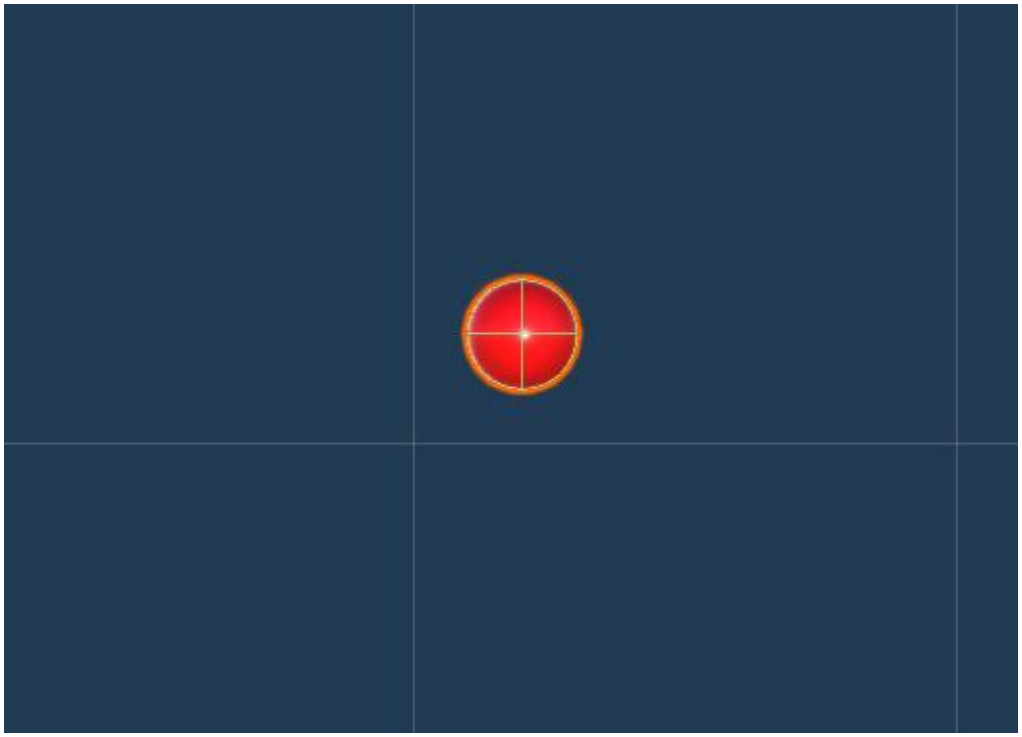

Figura 29 Modelo proyectil básico del jugador

Como ya se ha explicado anteriormente todo lo relacionado con las estadísticas del jugador debe de comunicarse con el sistema de objetos para ver dichas estadísticas aumentadas de manera correcta según el objeto que se recoja.

La implementación de recibir daño por parte de los enemigos gracias al uso del componente **Collider** en el jugador y del uso de las etiquetas como ya se ha explicado anteriormente en otras secciones, donde en este caso se deberían tener en cuenta las etiquetas que usan los diferentes enemigos desarrollados o los proyectiles que pueden llegar a disparar.

Por último, a la hora de detectar las colisiones con los enemigos y recibir daño, por cada daño que recibe el jugador, se comprueba si este mismo debe de ser eliminado, es decir, si la salud del jugador es igual a cero o menor, en caso afirmativo se pasaría a la escena final del juego dónde se le permite al jugador volver a jugar de nuevo o salir del juego.

El modelo y la estructura completa del jugador con el componente de **Collider** puede verse en la Figura 30.

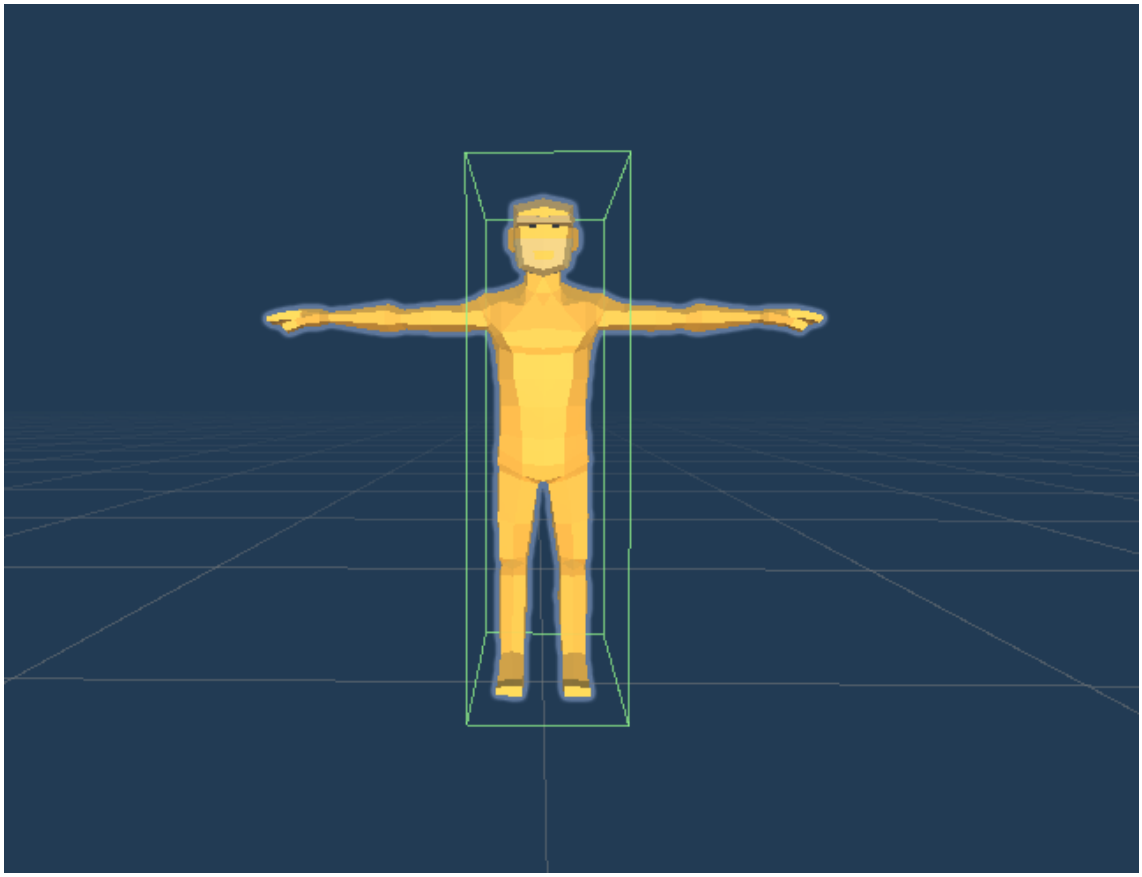

Figura 30 Estructura y modelo del jugador

5.6 Desarrollo del sistema de UI

Para el desarrollo del sistema de UI, hay que tener en constancia que este mismos depende completamente tanto del sistema de generación de niveles como del sistema de jugador, es decir, son estos dos últimos dos sistemas los encargados de enviar la información al sistema de la UI para que este mismo sea el encargado de generar la información relevante con respecto a lo que se envía, pero en la interfaz de usuario y con el formato correspondiente.

Debido a esto por cada habitación nueva generada desde el sistema de generación de niveles este sistema de UI es notificado con el tipo de habitación que se ha creado. Esto es bastante importante que las habitaciones tengan el mismo orden a la hora de especificarlas debido a que todas las habitaciones del mismo tipo ocupan la misma posición con respecto a la estructura de datos en las que se le definen, tanto para el sistema de UI como para el sistema de generación de niveles.

De esta forma cada vez se genera un nuevo nivel, o al ejecutar el juego por primera vez, siempre se genera la primera habitación que es la habitación de entrada que se utiliza como plantilla durante la generación de los niveles, pero en este caso con la plantilla para la interfaz de usuario. Esta habitación al igual que en la generación de los niveles con

la generación de nuevas habitaciones se establece en la posición 0,0,0. Es decir, la habitación se establece en la posición central del panel al que se añaden las habitaciones.

En este caso, todas las habitaciones se añaden a un *gameobject* vacío denominado como panel para que, al mover dicho panel, las referencias de las posiciones de todas las habitaciones generadas en el minimapa se muevan de la misma forma.

Esto se realiza para evitar que la generación de todas las habitaciones generadas por el sistema de generación de niveles haga que las habitaciones generadas en el minimapa se salgan de la propia interfaz de usuario por lo que en cada momento que se añade una nueva habitación al minimapa se comprueba si dicha habitación supera los límites de la interfaz de usuario para que no se quede ninguna habitación del minimapa fuera de la propia interfaz.

Este límite que se establece con un valor predefinido se ve aumentando según el número de habitaciones que se vayan generando fuera del mismo límite, esto se realiza para que el límite sea variable y no estático, y evitar que se tengan en cuenta habitaciones que ya se han comprobado anteriormente que no supere el límite y de esta forma conseguir que solo se comprueben los límites para las nuevas habitaciones que se van generando en el minimapa. De esta forma se consigue tratar los dos límites de manera independiente, teniendo en cuenta cuantas habitaciones se han pasado de cada límite, implementando tanto el límite superior como el límite derecho de la interfaz de usuario.

La forma en la que se generan las habitaciones en el minimapa, así como, la forma en la que se guarda la referencia de las habitaciones que se van creado en el minimapa se realiza de la misma forma que se realiza en la generación de niveles que ya se ha explicado anteriormente, es decir, la habitación se genera dependiendo del tipo de habitación que se requiere en dicha posición y se almacena en una estructura de datos igual que la explicada anteriormente con las coordenadas tanto del eje x como del eje z.

Uno de los ejemplos de las diferentes habitaciones que se generan en el minimapa a través de las diferentes plantillas de las diferentes habitaciones que se generan en el sistema de generación de niveles, pero con un cambio de tipo a un **Sprite2D**. El diseño de este objeto ha sido obtenido de un paquete de **Assets** gratuito especificado en la **Asset Store** de **Unity** [4] para la habitación principal como plantilla y se han realizado a mano las diferentes habitaciones a partir de esta. Una de las plantillas que se ha utilizado para las habitaciones se puede ver en la Figura 31.

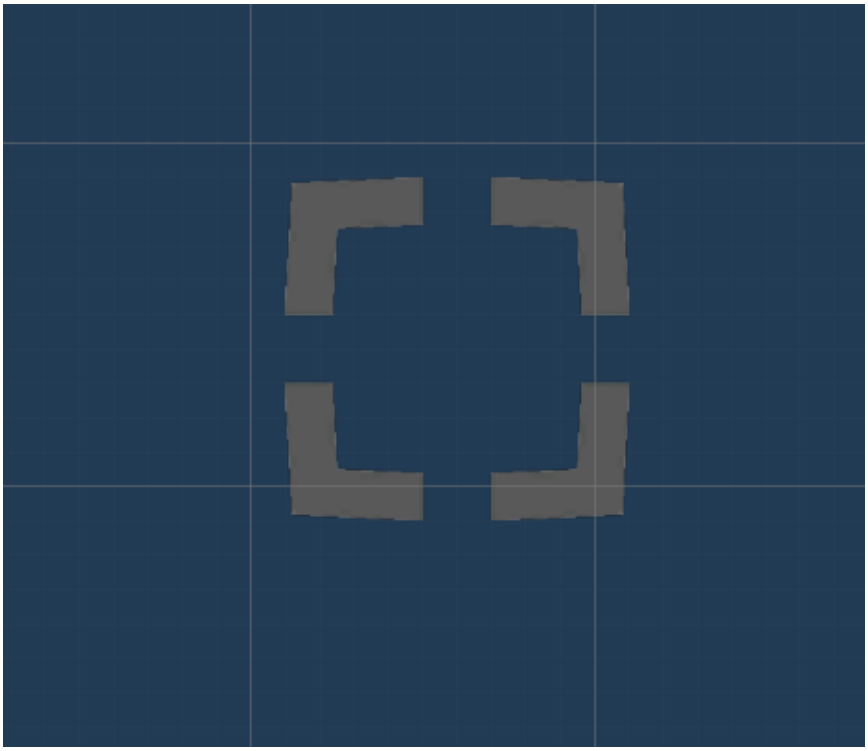

Figura 31 Plantilla de habitación principal del minimapa

Por otro lado, para la implementación de la cantidad de salud que le queda al jugador en todo momento, se ha realizado uno cálculo especial para contar con dos tipos de textura según la cantidad de salud que le quede al jugador.

Para la implementación de la salud en la interfaz de usuario del jugador, se han implementado las texturas de un corazón y un corazón roto representando a un valor medio (por ejemplo 50.5 de salud actualmente). El diseño del modelo específicamente del corazón ha sido obtenido de un paquete de **Assets** gratuito especificado en la **Asset Store** de **Unity** [4] y el modelo para el corazón roto se ha realizado a mano a partir del modelo del corazón completo. Ambos modelos de estos dos diseños de los corazones pueden verse en la Figura 32 y Figura 33.

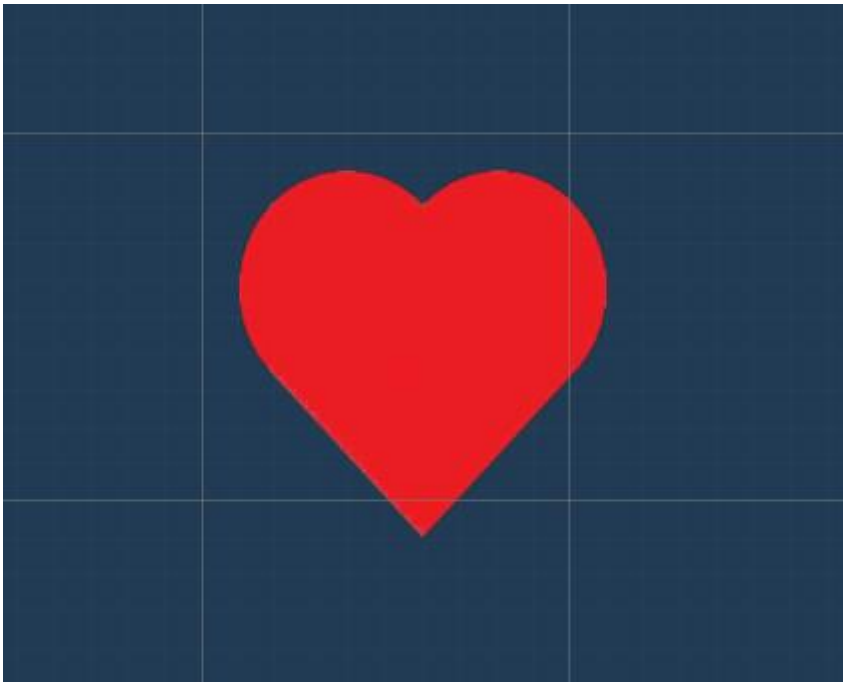

Figura 32 Modelo Sprite2D de corazón

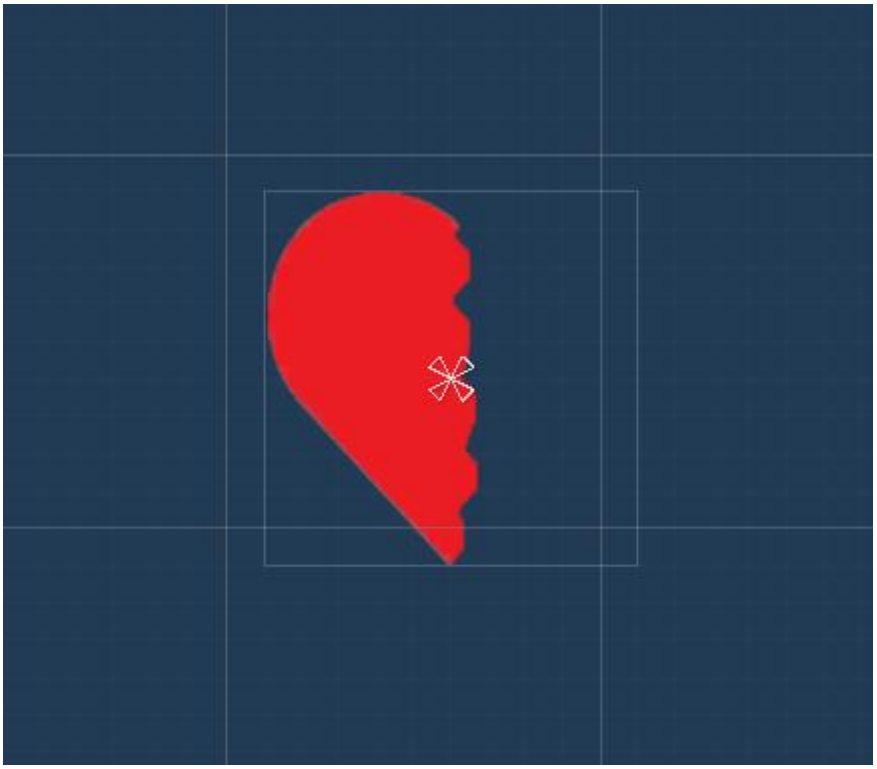

Figura 33 Modelo Sprite2D de corazón roto

Por último, con respecto a la implementación del sistema de UI, solo queda explicar el desarrollo e implementación del puntero que representa al jugador en el propio minimapa.

Este puntero que indica dicha posición se actualiza desde el movimiento realizado tanto para el jugador como para la cámara desde el *GameManager*.

En este caso el sistema de UI comprueba si la habitación de dicha posición había sido descubierta anteriormente, y en caso contrario, la habilita para que el jugador pueda verla en la interfaz de usuario y por último se encarga de actualizar la posición del jugador en el minimapa, a través de actualizar la referencia del punto que se usa como indicación de esta.

El modelo de **Sprite2D** que se ha utilizado para indicar la posición del jugador en el minimapa se ha realizado a mano y puede verse en la Figura 34.

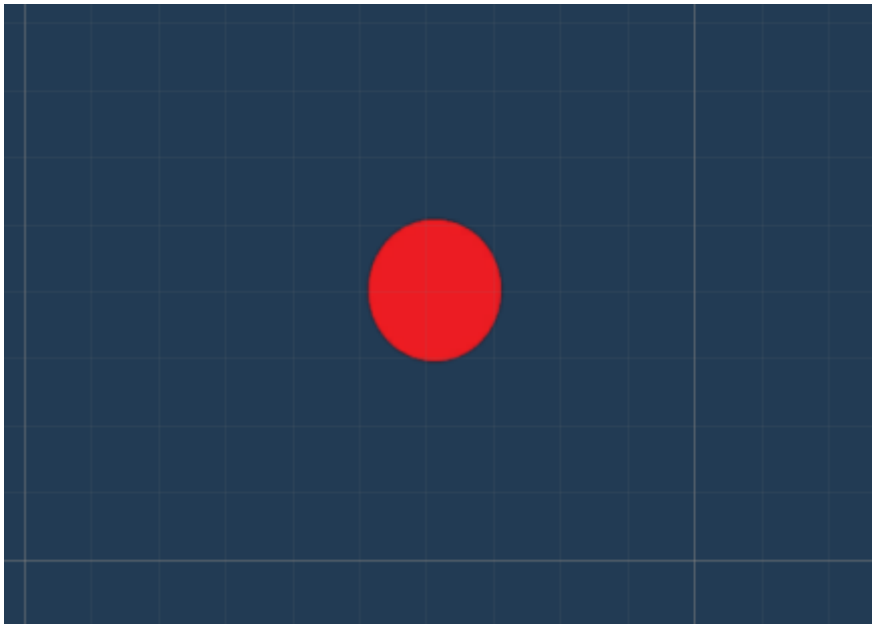

Figura 34 Modelo Sprite2D del indicador de posición del jugador en el minimapa

6. Manual de usuario

En este capítulo se van a exponer las instrucciones para usar el sistema desarrollado en este proyecto.

Para empezar a jugar, simplemente se tiene que ejecutar el archivo de aplicación **.exe**. Una vez se ejecuta el juego, el jugador aparece en medio de la primera sala del primer nivel y en este mismo instante se comenzaría a generar todo el nivel actual.

6.1 Controles del jugador

El movimiento del jugador se basa en el uso de las teclas "**W, A, S, D**" para mover el personaje por el nivel, utilizando la tecla "**W**" para el movimiento hacia arriba, la tecla "**S**" para el movimiento hacia abajo, la tecla "**D**" para el movimiento hacia la derecha y la tecla "**A**" para el movimiento hacia la izquierda.

Por otro lado, con respecto al movimiento, también se le otorga la posibilidad de disparar al jugador y para ello se utilizan las diferentes teclas de dirección del teclado quedando de la siguiente forma:

- Flecha hacia arriba: Dispara hacia arriba girando al personaje en dicha dirección.
- Flecha hacia abajo: Dispara hacia abajo girando al personaje en dicha dirección.
- Flecha hacia la derecha: Dispara hacia la derecha girando al personaje en dicha dirección.
- Flecha hacia la izquierda: Dispara hacia la izquierda girando al personaje en dicha dirección.

Por último, con respecto a la interacción por parte del jugador con los enemigos y los objetos se realiza de manera automática y no necesita interacción por parte del usuario que no implique moverse o disparar y en el momento de que el jugador muere se le establece la pantalla de fin del juego dónde se le da la posibilidad de jugar de nuevo o salir donde se elige dicha acción mediante el botón izquierdo del ratón.

7. Conclusiones

En este TFM se ha desarrollado un videojuego del subgénero "*roguelike*" en el cual los diferentes niveles se generan aleatoriamente a través de la elección aleatoria de habitaciones prediseñadas. Durante el desarrollo de este proyecto, se ha aprendido sobre todo la importancia que tiene la forma de generar el nivel en este tipo de juegos teniendo el foco en la lógica de cerrar el nivel y que no quede un nivel incompleto además del rendimiento que se necesita para generar el nivel en tiempo de ejecución mientras el jugador está ejecutando el videojuego.

Debido a esta dificultad con respecto a la generación aleatoria del nivel y todo lo que ello conlleva (generar los enemigos aleatoriamente, escoger la habitación del *boss, eliminar y generar de nuevo todo lo necesario al pasar a un nuevo nivel, etc.*.) durante el tiempo de desarrollo, no se ha logrado cumplir todos los objetivos que se querían cumplir desde un principio, dejando de lado la implementación de una historia para dar un sentido al juego para poder acabar a tiempo el desarrollo del videojuego con las bases que se han considerado más importantes.

Esto último nos lleva al incumplimiento de la planificación, es decir, como se ha explicado anteriormente en la planificación se incluyó el diseño e implementación de una historia, pero por problemas de tiempo en el desarrollo se ha decidido prescindir de esta parte porque no se considera como una parte base del videojuego y por lo tanto se podría considerar como el único cambio que ha sufrido la planificación a lo largo del desarrollo del proyecto.

6.1 Trabajo Futuro

A continuación, se indican algunas funcionalidades que podrían añadirse o modificarse como trabajo futuro:

- Modificar la generación de niveles para independizarla de los fotogramas y que se realice de una manera más eficiente utilizando, por ejemplo, las corrutinas que nos ofrece Unity y que causaría una gran mejora en tema de rendimiento a la hora de generar los niveles eligiendo las diferentes habitaciones. Este cambio podría resultar en un cambio completo en la forma en la que se generan los niveles en este momento, por lo que habría que replanteare modificar los diferentes algoritmos desarrollados para generar los niveles e incluso las formas de las habitaciones prediseñadas.
- Además de esto, también sería bastante interesante diseñar e implementar tanto más habitaciones como más *bosses* para añadir muchísima más aleatoriedad a la generación de niveles y más diversidad al juego.

- Por otro lado, sería muy interesante añadir sonido al videojuego, añadiendo una melodía durante el transcurso del juego, o durante los enfrentamientos contra los *bosses.*
- Además, con respecto al tema de implementación de la historia, sería muy importante añadir e implementar una historia durante el transcurso del juego y así cumplir con todos los objetivos que se pretendían cumplir desde un principio.
- Y, por último, sería muy interesante dotar al videojuego de un menú de opciones y menú principal tanto a la hora de inicia el juego, como durante el transcurso de este.

8. Glosario

- GameManager: Script que mantiene el estado actual del videojuego en ejecución.
- Asset [16]: Objeto creado que puede usarse en Unity3D y puede provenir de un archivo creado fuera de Unity3D como, por ejemplo, un modelo 3D, un audio, etc…
- Minimapa: Mapa en miniatura que suele situarse en una esquina de la pantalla y su objetivo es ayudar al jugador a orientarse dentro del videojuego.
- Gameobject [14]: Objeto fundamental de Unity3D que puede representar diferentes entidades en el motor de videojuegos.
- Collider [15]: Componente que define la forma de un objeto para los propósitos de colisiones físicas.
- Prefab [12]: Este concepto se puede definir como una plantilla de objeto utilizada en Unity3D.
- Boss: Concepto utilizando en el mundo de los videojuegos para definir a un enemigo especial, normalmente más difícil de eliminar que los demás.
- NavMeshAgent [13]: Componente que se utiliza para encontrar caminos mediante algoritmo a lo largo de un espacio tridimensional.
- Engine (motor de videojuegos): Un engine en este contexto, se trata de un motor de videojuegos, es decir, una plataforma que permite desarrollar un videojuego.

9. Bibliografía

[1] POLYGON Prototype - Low Poly 3D Art by Synty: https://assetstore.unity.com/packages/3d/props/exterior/polygon-prototype-low-poly-3d-art-by-synty-137126 (01/06/2021)
[2] Mixamo: https://www.mixamo.com/#/ (01/06/2021)
[3] UMotion Pro – Animation Editor: https://assetstore.unity.com/packages/tools/animation/umotion-pro-animation-editor-95991 (01/06/2021)
[4] 40+ Simple Icons - Free: https://assetstore.unity.com/packages/2d/gui/icons/40-simple-icons-free-171008 (01/06/2021)
[5] 3D Game Kit – Character Pack: https://assetstore.unity.com/packages/3d/3d-game-kit-character-pack-135217 (01/06/2021)
[6] Unity Real-Time Development: https://unity.com/ (01/06/2021)
[7] Unreal Engine: https://www.unrealengine.com/en-US/ (01/06/2021)
[8] Godot Engine: https://godotengine.org/ (01/06/2021)
[9] Loop Hero: https://loophero.com/ (01/06/2021)
[10] Hades: https://www.supergiantgames.com/games/hades/ (01/06/2021)
[11] The Binding of Isaac: https://store.steampowered.com/app/113200/The_Binding_of_Isaac/ (01/06/2021)
[12] Unity Prefabs: https://docs.unity3d.com/es/2021.1/Manual/Prefabs.html (02/06/2021)
[13] Unity NavMesh Agent: https://docs.unity3d.com/es/2021.1/Manual/class-NavMeshAgent.html (02/06/2021)
[14] Unity GameObject: https://docs.unity3d.com/es/2021.1/Manual/GameObjects.html (02/06/2021)
[15] Unity Colliders: https://docs.unity3d.com/es/2021.1/Manual/CollidersOverview.html (02/06/2021)
[16] Quick guide to Unity Asset Store: https://unity3d.com/quick-guide-to-unity-asset-store (02/06/2021)
[17] Requisitos Unity: https://unity3d.com/es/get-unity/download (02/06/2021)