



# UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

## Diseño de un motor de videojuegos 3D para el estudio de avatares virtuales inteligentes

Realizado por  
**Jesús López Pujazón**

Para la obtención del título de  
Doble Grado en Ingeniería Informática y Matemáticas

Dirigido por  
Pablo García Sánchez  
Carlos Ureña Almagro

Realizado en el departamento de  
ATC y LSI

Convocatoria de Junio, curso 2021/22

---

# Agradecimientos

---

Quiero agradecer a X por...

También quiero agradecer a Y por...

---

# Índice general

---

<b>1. Resumen</b>	<b>1</b>
<b>2. Abstract</b>	<b>2</b>
<b>3. Introducción</b>	<b>3</b>
3.1. Contexto del proyecto . . . . .	3
3.2. Problema abordado . . . . .	3
3.3. Técnicas y conceptos empleados . . . . .	3
<b>4. Objetivos</b>	<b>4</b>
4.1. Introducción . . . . .	4
4.2. Objetivos . . . . .	4
<b>5. Desarrollo del trabajo</b>	<b>5</b>
5.1. Análisis del problema . . . . .	5
5.1.1. Introducción . . . . .	5
5.1.2. Planificación y presupuesto . . . . .	5
5.1.3. Requerimientos . . . . .	6
5.1.4. Herramientas . . . . .	8
5.1.5. Modelado . . . . .	9
5.2. Implementación . . . . .	11
5.2.1. Introducción . . . . .	11
5.2.2. Diseño de la escena . . . . .	12
5.2.3. Implementación del algoritmo . . . . .	14
5.2.4. Prueba inicial . . . . .	16
5.3. Pruebas . . . . .	22
5.3.1. Introducción . . . . .	22
5.3.2. Conclusiones . . . . .	22
<b>6. Conclusiones y vías futuras</b>	<b>23</b>
<b>7. Bibliografía</b>	<b>24</b>

---

# Índice de figuras

---

5.1.	Ciclo de vida del enemigo . . . . .	7
5.2.	Boceto del escenario de la aplicación . . . . .	7
5.3.	Herramientas empleadas . . . . .	9
5.4.	Boceto y modelo del enemigo básico . . . . .	10
5.5.	Enemigo básico: Modelo renderizado . . . . .	10
5.6.	Definición del esqueleto del modelo . . . . .	11
5.7.	Estructuras: Modelos renderizados . . . . .	11
5.8.	Tipos de proyecto disponibles en Unity . . . . .	12
5.9.	Orden de ejecución en Unity . . . . .	13
5.10.	Escenario básico para el desarrollo del proyecto . . . . .	14
5.11.	Escenario final para el desarrollo del proyecto . . . . .	16
5.12.	Entrenamiento de la IA en múltiples escenarios . . . . .	18
5.13.	Variables en el editor de Unity . . . . .	19

---

# Índice de extractos de código

---

5.1.	FortalezaHealth.cs	14
5.2.	Crear un entorno virtual con python	15
5.3.	Instalación del paquete pytorch y mlagents	16
5.4.	Versión inicial del algoritmo	17
5.5.	Segunda versión del algoritmo para atacar la fortaleza	18
5.6.	Añadidos a GoToGoal para indicar la salud el enemigo	20
5.7.	Algoritmo de la torreta	21

---

# 1. Resumen

---

A lo largo de la primera década del siglo XXI surgieron diversas videoconsolas que normalizarían la presencia de estas en cualquier hogar. Desde entonces, la industria del videojuego ha crecido de manera casi exponencial en importancia hasta alcanzar una increíble fama e influencia en todo el mundo. Uno de los aspectos más importantes en la mayoría de videojuegos se trata de la Inteligencia Artificial que poseen los NPC (Non Playable Characters). En ocasiones, juegos cuyos conceptos podrían tener un gran potencial, se ven eclipsados por la presencia de enemigos cuya inteligencia les obliga a perseguir de forma patosa al personaje que controlamos. En este trabajo, modelaré algunas estructuras y personajes mediante la técnica de *Voxel Art* y aplicaré animaciones a los personajes en una web definiendo sus articulaciones principales para dar lugar a un escenario de un videojuego. Asimismo, aplicaré técnicas de aprendizaje por refuerzo a una inteligencia artificial para comprobar si se podrían aplicar estas técnicas en la industria para dar lugar a una experiencia de juego más rica. El objetivo de este trabajo será, por tanto, estudiar la eficiencia de algoritmos de aprendizaje por refuerzo en un caso real para comprobar su eficacia.

**Palabras clave:** Inteligencia Artificial, NPC, Voxel Art, Aprendizaje por refuerzo, Modelar.

---

## 2. Abstract

---

The first decade of the 21st century saw the emergence of various video game consoles that would normalise their presence in every home. Since then, the video game industry has grown almost exponentially in importance to achieve incredible fame and influence around the world. One of the most important aspects of most video games is the Artificial Intelligence that NPCs (Non Playable Characters) possess. Sometimes, games whose concepts could have great potential, are overshadowed by the presence of enemies whose intelligence forces them to chase after the character we control in a clumsy way. In this paper, I will model some structures and characters using the technique of *Voxel Art* and I will apply animations to the characters in a web defining their main articulations to create a video game scenario. I will also apply reinforcement learning techniques to an artificial intelligence to see if these techniques could be applied in the industry to create a richer gaming experience. The aim of this work will therefore be to study the efficiency of reinforcement learning algorithms in a real case to test their effectiveness.

**Keywords:** Artificial Intelligence, NPC, Voxel Art, Reinforcement Learning, Model.

---

## 3. Introducción

---

### 3.1. Contexto del proyecto

Desde la creación de los videojuegos en la década de los 50, la industria ha experimentado un crecimiento constante, dando lugar a una de las industrias más importantes y poderosas en el sector del ocio. Desde sus inicios los videojuegos y la Inteligencia Artificial (IA) han estado estrechamente ligados, de hecho, algunos de los primeros juegos, consistentes en juegos de ajedrez por ordenador, ya mostraban una inteligencia capaz de elegir la mejor jugada posible basándose en las posiciones de las piezas en el tablero. Surge así, con los primeros videojuegos, la necesidad de avanzar y profundizar en el desarrollo de mejores algoritmos para crear enemigos más inteligentes, siendo Pacman, por ejemplo, uno de los primeros en el que los enemigos cuentan con un sistema de búsqueda de rutas. Junto con la aparición de enemigos más inteligentes empiezan a surgir nuevas consolas más potentes que permiten generar cada vez gráficos más realistas. Sin embargo, a pesar de que algunas de las consolas de última generación son capaces de cargar gráficos fotorrealistas en tiempos de carga ínfimos, estos coexisten con distintos estilos artísticos que surgieron a lo largo de los años. Un ejemplo de esto son los videojuegos con un estilo artístico de pixel art, en los que las imágenes se editan al nivel del píxel, o videojuegos como *Minecraft*, que a pesar de emplear un estilo artístico basado en el empleo de cubos, se ha convertido en el juego más vendido de la historia, seguido del popular juego *Tetris*.

En la actualidad coexisten distintos tipos de videojuegos, con escenarios en dos o tres dimensiones, en los que la variedad de combinaciones es abrumadora y en los que cada vez resulta más complicado generar algoritmos para la IA que controla a los enemigos que se muestran por pantalla. Para ofrecer una jugabilidad más rica y que permita a los jugadores tener una experiencia de juego única, algunos desarrolladores, como *Dynamic Pixels* [2], están trabajando con nuevos patrones de IA que permitan al enemigo observar y aprender de las acciones del jugador para actuar según el método de juego de cada uno.

### 3.2. Problema abordado

### 3.3. Técnicas y conceptos empleados

---

## **4. Objetivos**

---

### **4.1. Introducción**

Breve introducción al capítulo

### **4.2. Objetivos**

---

# 5. Desarrollo del trabajo

---

## 5.1. Análisis del problema

### 5.1.1. Introducción

En este apartado comentaré la planificación que seguiré para el desarrollo de este proyecto, así como el presupuesto estimado necesario para su creación, las herramientas a emplear durante el desarrollo y los requerimientos y metodología de desarrollo por los que he optado.

### 5.1.2. Planificación y presupuesto

Dado que el primer cuatrimestre presenta una mayor carga respecto al número de asignaturas, prácticas y exámenes, dedicaré este cuatrimestre a organizar el desarrollo del proyecto, estudiar las distintas alternativas que se presentan para realizar el desarrollo de este, establecer los primeros pasos a realizar con ambos tutores y preparar todas las herramientas necesarias que se decidan para poder empezar a realizar el proyecto durante el segundo cuatrimestre.

**Planificación del primer semestre:** para la primera quincena de Octubre deberé haber contactado con ambos tutores para informarme de cuáles son los primeros pasos a tomar durante el desarrollo del proyecto, qué horarios tienen disponibles para preguntarles dudas y qué herramientas recomiendan utilizar para la implementación de este. Para finales de Noviembre necesito haber decidido qué herramientas utilizar para el desarrollo del TFG e instalarlas para comprobar que funcionan correctamente y, si hubiese algún problema relacionado con esto, contactar lo antes posible con los tutores para intentar solucionar los problemas que surjan. El mes de Diciembre estará disponible para resolver esos problemas y poder centrarme en los exámenes del semestre.

La planificación del primer cuatrimestre se ha cumplido según lo previsto, he decidido las herramientas a emplear para el desarrollo del TFG, a excepción del método para programar el algoritmo de aprendizaje por refuerzo, y he instalado todas las herramientas necesarias, comprobado su funcionamiento, y realizado algunas pruebas con estas para una primera toma de contacto.

**Planificación del segundo semestre:** tras el período de examenes finales del anterior cuatrimestre empezaré a realizar bocetos del personaje principal que actuará en la escena y las estructuras que se encontrarán presentes en esta. Durante el mes de Marzo, modelaré a este personaje y estructuras. Además crearé alguna animación para el personaje y la importaré a la plataforma de desarrollo del videojuego para comprobar que funciona correctamente. Los meses de Abril y Mayo los dedicaré exclusivamente a

desarrollar el videojuego y el algoritmo de aprendizaje por refuerzo, así como desarrollar la memoria del trabajo. Para finales de Abril necesitaré tener una versión básica funcional (al menos) del algoritmo de aprendizaje por refuerzo para poder comentar el progreso y desarrollo de este con el tutor correspondiente.

A diferencia del primer cuatrimestre, esta planificación no ha sido tan sencilla de seguir debido a retrasos en la realización de algunas de las actividades, como el desarrollo de la memoria, que se atrasó debido a dificultades para conseguir ejecutar la versión inicial del algoritmo de aprendizaje por refuerzo. Esto, añadido a una masiva aparición de prácticas entregables de otras asignaturas y exámenes parciales provocó un retraso de una semana y media aproximadamente en el desarrollo de las actividades previstas.

Para el desarrollo de este videojuego será necesario, al menos, un ordenador portátil o de sobremesa para poder modelar e implementar las estructuras de la escena y el algoritmo de aprendizaje por refuerzo respectivamente. En mi caso, emplearé mi ordenador portátil, en concreto un MSI modelo GE72 7RE, valorado en aproximadamente 900€. Además, habría que añadir el sueldo de un programador, que rondaría entorno a los 20€/hora. Como el TFG consta de 18 créditos, le corresponden aproximadamente unas 450 horas de trabajo. Por tanto, suponiendo un pago como el anterior y estas horas el precio total por las horas de trabajo ascendería a 9.000€. En este caso, el software empleado es totalmente gratuito, por lo que no se tendrá en cuenta el coste de una posible licencia más avanzada. Por tanto, el precio total sería, teniendo en cuenta el hardware y las horas de trabajo de un sólo programador, de 9.900€.

### 5.1.3. Requerimientos

A continuación realizo un análisis de los requisitos funcionales y no funcionales del proyecto, así como de otros requerimientos generales necesarios para completar con éxito el desarrollo de este.

**Requisitos funcionales:** el enemigo básico debe ser capaz de desplazarse por todo el mapa dentro de los límites establecidos, volviendo a su posición inicial en el caso de que sobrepase los límites del escenario. Además el enemigo debe poder atacar tanto el almacén de oro, consiguiendo dinero cuando realize esta acción, y debe poder atacar la fortaleza, cuyo objetivo será reducir su salud hasta el mínimo posible. Si el enemigo consigue recolectar todo el dinero del almacén de oro, este obtendrá un bonus de ataque hasta el final de la partida, que le permitirá realizar el doble de daño a la fortaleza. El enemigo no podrá interactuar de forma directa con la torreta con el objetivo de que este aprenda a evitarla en lugar de intentar destruirla. La torreta deberá apuntar constantemente al enemigo, infligiéndole constantemente daño mediante disparos. El daño que realizará la torreta al enemigo dependerá de la distancia a la que se encuentren ambos elementos. La fortaleza y el almacén de oro no podrán interactuar con ningún otro elemento de la escena.

El ciclo de vida del enemigo puede observarse en la figura 5.1.

**Requisitos no funcionales:** la aplicación será una aplicación ejecutable en PC desde el escritorio. El framerate de la aplicación se mantendrá estable y nunca deberá

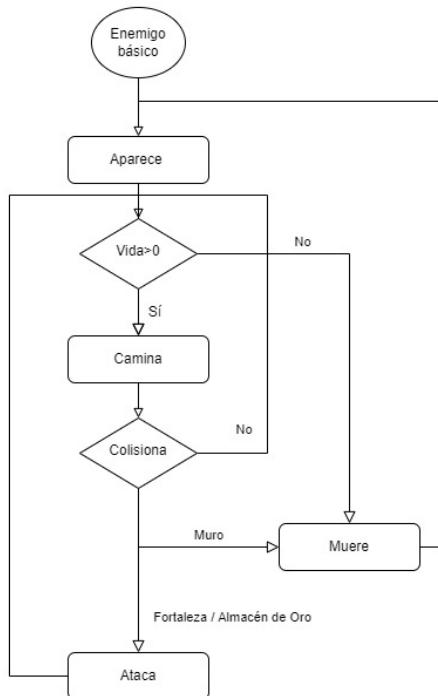


Figura 5.1: Ciclo de vida del enemigo

ser inferior a los 30 FPS (Frames Por Segundo).

El proyecto constará de un sólo nivel, aunque las posiciones en las que se generen ciertos elementos de este como la torreta o el almacén de oro serán pseudoaleatorias. La interfaz mostrará además, una barra de salud que mostrará el estado en el que se encuentra la salud de la fortaleza principal y una barra que mostrará la cantidad de dinero que queda en el alamcén de oro, para poder comprobar fácilmente si el enemigo ha conseguido robar todo el dinero de ésta. A continuación muestro un boceto de la escena:

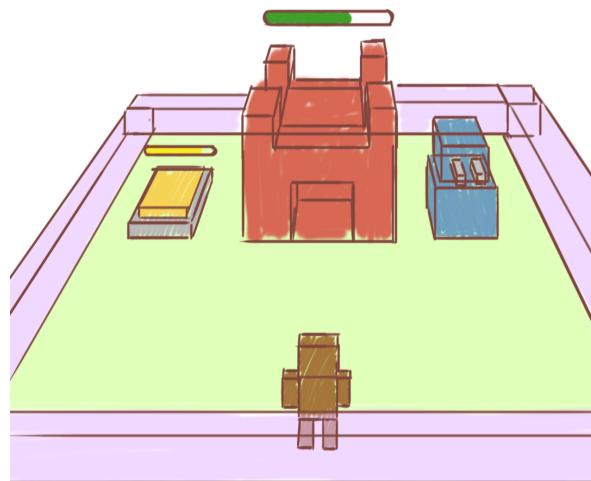


Figura 5.2: Boceto del escenario de la aplicación

#### 5.1.4. Herramientas

Para el desarrollo del proyecto ha sido necesario el uso de distintas herramientas. Para el modelado de las estructuras y los personajes he utilizado el software gratuito **MagicaVoxel** [3], que permite modelar mediante el empleo de voxels, así como renderizar los objetos creados y aplicarles distintas texturas. Entre los distintos programas de modelado existentes me he decantado por MagicaVoxel por diversos motivos. El primer motivo ha sido mi familiaridad con el uso de este, lo que agilizará el diseño de los modelos sin tener que invertir demasiado tiempo en aprender a usar un nuevo programa de modelado. Además, el uso de voxels permite representar secillamente cualquier sólido de manera aproximada. Otros programas, como Blender, permiten generar modelos más definidos con un esqueleto más complejo, sin embargo, como utilizaré una web para generar las animaciones de forma más sencilla no será necesario dedicar mucho tiempo a definir un esqueleto con mayor complejidad.

Para crear las animaciones de los personajes he empleado la página **Mixamo** [1], que permite generar distintas animaciones predeterminadas para los esqueletos de los personajes y, posteriormente, exportarlas de manera eficiente para poder utilizarlas en otros software como Unity, Blender o Unreal Engine. Esta web me permitirá generar cualquier animación que necesite sin necesidad de crear un esqueleto muy complejo para los personajes, ya que proporciona un método para indicar dónde se encontrarían las articulaciones de los modelos que se importan y, a partir de esos puntos, genera la animación.

Para diseñar la escena y poder implementar el videojuego junto con el algoritmo de Aprendizaje por Refuerzo he empleado **Unity** [4], que permite la creación de escenas y videojuegos tanto 2D como 3D y posee diversos tutoriales y ayudas para sus usuarios. Unity es una de las aplicaciones más usadas por los desarrolladores de videojuegos tanto principiantes como profesionales. Existen otras aplicaciones muy usadas, como Unreal Engine, sin embargo estas suelen emplearse en desarrollos con gráficos fotorrealistas ya que están enfocadas más en ese tipo de proyectos. Por esto mismo, este entorno de desarrollo me permitirá desarrollar el proyecto en menos tiempo que si lo programase a bajo nivel empleando, por ejemplo, una especificación como OpenGL. Una de las ventajas respecto a esta reside en la facilidad que presenta Unity para crear objetos y figuras básicas como cilindros, planos, etc. sin necesidad de definir sus vértices y caras y pudiendo modificar sus parámetros de manera rápida y sencilla. Además, Unity incorpora la posibilidad de añadir cámaras y fuentes de luz con tan sólo un clic y contiene diversos paquetes que pueden instalarse de manera completamente gratuita facilitando el desarrollo de cualquier proyecto. Por último, otra de las razones más importantes por las que he lo he escogido, reside en la facilidad para encontrar ayuda en la web, siendo numerosos los tutoriales y ayudas que se pueden encontrar de manera casi instantánea.

Por último, para implementar el algoritmo, utilizaré **ML-Agents** [5]. ML-Agents (Unity Machine Learning Agents Toolkit), un proyecto open-source que se puede instalar en Unity como un paquete que permite crear avatares inteligentes y entornos complejos para el entrenamiento de los modelos. Los agentes de ML de Unity se integran como un paquete, permiten realizar entrenamientos conectando el proyecto integrado y entrenando a los agentes para que aprendan y, por último, insertar el modelo del agente

entrenado en el proyecto de Unity. Entre las principales ventajas de este paquete, como se comenta en la web oficial de Unity [6], residen que se trata de una herramienta de código abierto, es fácil de configurar sin necesidad de tener mucha experiencia en el campo, lo que me permitirá avanzar más rápido en el desarrollo sin tener que dedicar mucho tiempo en detalles a bajo nivel y centrándome, por tanto, en el algoritmo de entrenamiento de la IA del enemigo. Además, al igual que comenté en las razones para usar Unity, existen numerosos tutoriales y ayudas online que me permitirán resolver más rápidamente los errores que se presenten durante el proceso de desarrollo.



Figura 5.3: Herramientas empleadas

### 5.1.5. Modelado

Para el desarrollo de un primer escenario sencillo será necesario, en primer lugar, modelar tanto las tres estructuras básicas que compondrán la escena y un enemigo básico. Para el modelado emplearé, como se ha comentado en el apartado anterior, el programa *MagicaVoxel*.

En primer lugar comenzaré modelando el enemigo básico que utilizaré durante todo el desarrollo del problema. Para ello emplearé una cuadrícula de tamaño 50x21x56. En este caso utilizaré una cuadrícula más grande que la que usaré para modelar los tres edificios principales ya que será necesario definir bien las partes del enemigo donde se encontrarán sus articulaciones (las muñecas, los codos y las rodillas). Para el diseño, me he basado en un boceto de un personaje que realicé hace tiempo y le he añadido algunas modificaciones. La principal modificación necesaria es eliminar voxels en las zonas donde se encuentran las articulaciones para poder detectarlas correctamente en *Mixamo*. El modelo del personaje obtenido es el siguiente:

A continuación, crearé una animación de movimiento para este enemigo utilizando la web *Mixamo*. Para ello, inicio sesión en la web y en la página principal encontramos un modelo de ejemplo para generar animaciones. Para poder generar la animación, cargo el modelo a partir de un fichero zip que contendrá el modelo del enemigo, las texturas y los materiales. Una vez cargado el modelo, es necesario indicar las posiciones de las articulaciones para poder generar la animación correctamente. A continuación se muestra el modelo con la barbilla, codos, muñecas, rodillas y cintura marcadas. Están representadas mediante un círculo de color azul, amarillo, verde, naranja y rosa respectivamente.

Para finalizar, es necesario escoger una animación entre las que ofrece la web, en este caso he escogido la animación *Mutant Walking*. Una vez finalizado, necesitaré

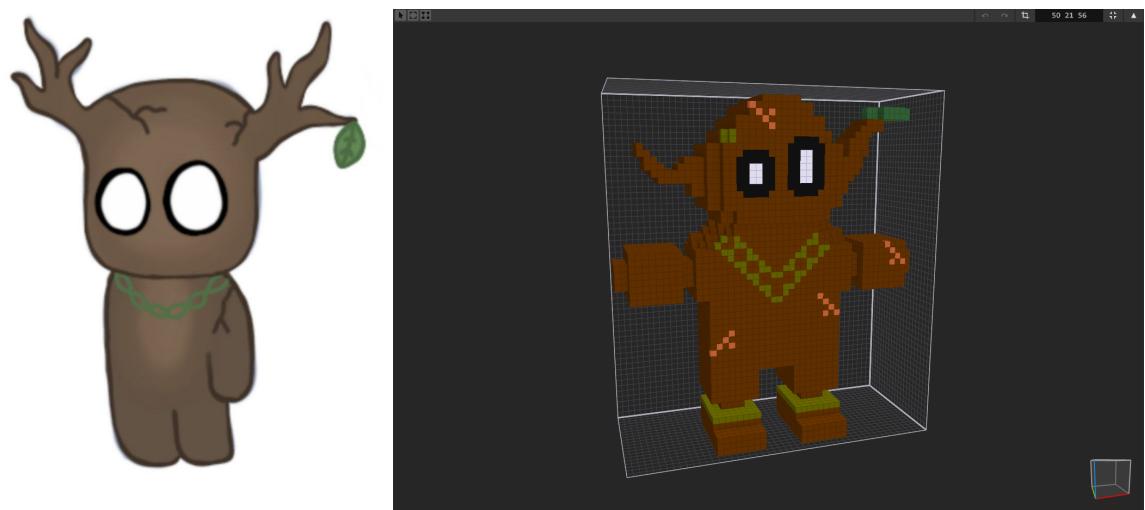


Figura 5.4: Boceto y modelo del enemigo básico



Figura 5.5: Enemigo básico: Modelo renderizado

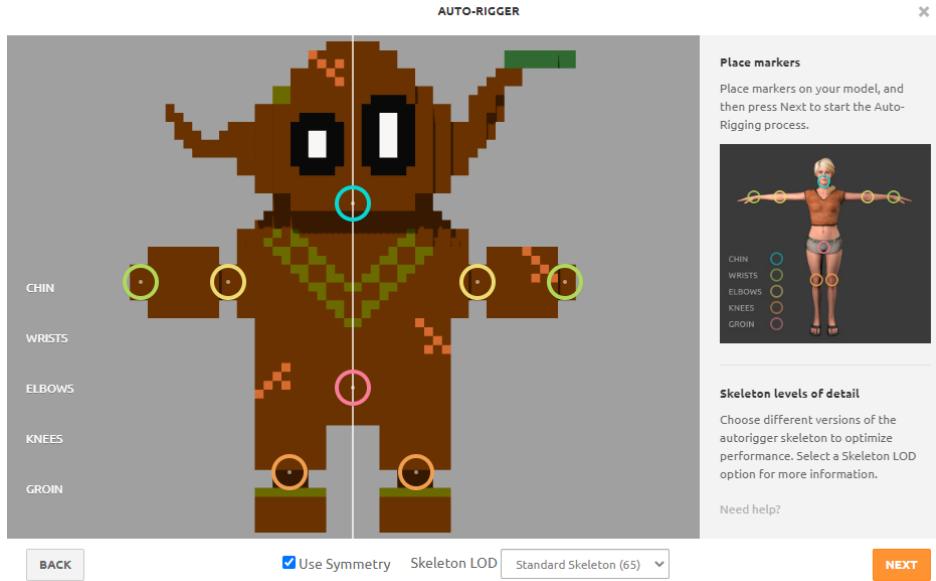


Figura 5.6: Definición del esqueleto del modelo

modelar las tres estructuras básicas que compondrán el escenario del juego. Estas estructuras son: la torreta, el almacén de oro y la fortaleza. Para modelar estas tres usaré una cuadrícula del mismo tamaño, en este caso una cuadrícula de tamaño 40x40x40. Para el diseño, esta vez me he inspirado en las estructuras del videojuego *Clash of Clans*. Los modelos renderizados de las estructuras pueden observarse a continuación.



Figura 5.7: Estructuras: Modelos renderizados

Para el modelado de la torreta, decidí separar la parte superior de la inferior para tener dos modelos diferenciados. De este modo la parte superior podrá girar cuando se desarrolle el videojuego sin necesidad de girar toda la estructura.

## 5.2. Implementación

### 5.2.1. Introducción

En este capítulo explicaré el procedimiento que he seguido para la creación de la escena del videojuego a partir de todos los componentes modelados y la implementación

del algoritmo de Aprendizaje por Refuerzo que utilizará la IA para aprender a atacar la fortaleza principal y maximizar el daño que esta reciba.

### 5.2.2. Diseño de la escena

Para el desarrollo restante del proyecto usaré la herramienta *Unity*. En primer lugar, Unity permite crear distintos tipos de proyectos los cuales se muestran a continuación:

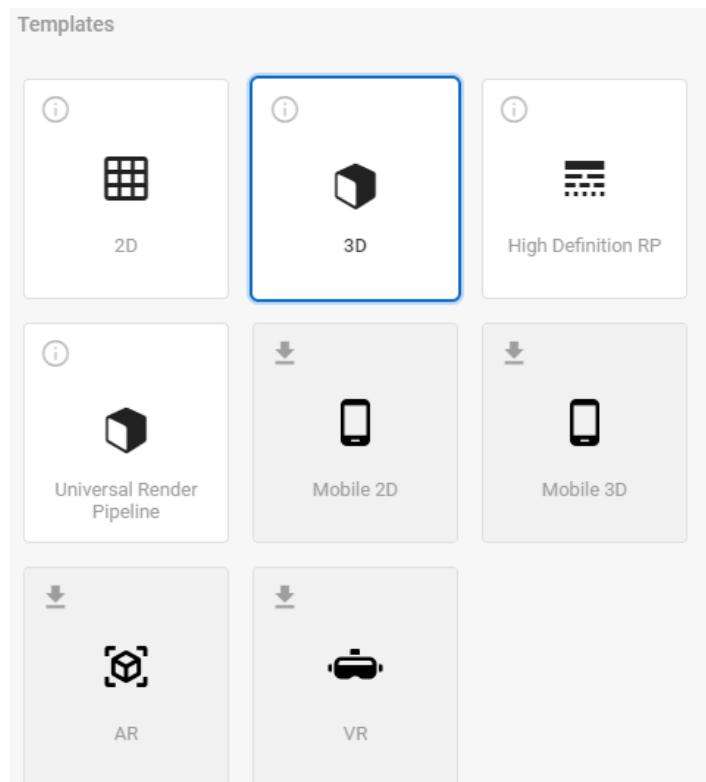


Figura 5.8: Tipos de proyecto disponibles en Unity

Como se observa en la imagen, Unity permite realizar una gran variedad de proyectos, entre estos, podemos encontrar proyectos en 2D como algunos videojuegos clásicos en los que los personajes y elementos se encuentran totalmente en dos dimensiones, proyectos en 3D, que permiten tanto crear videojuegos (como haré en este proyecto) como crear animaciones y escenarios en tres dimensiones. Se encuentran además otras alternativas como High Definition RP, que se emplean para proyectos en los que la iluminación juega un papel importante; Universal Render Pipeline, que está enfocado en proyectos que buscan representar gráficos más realistas; aplicaciones en 2D y 3D para móviles, aplicaciones en realidad aumentada (AR) y realidad virtual (VR).

En este caso elijo crear un proyecto 3D, para ello sólo es necesario introducir el nombre del proyecto y el directorio donde se creará. Una vez creado, el editor de Unity genera automáticamente una escena que contiene únicamente una cámara y una fuente de luz. Como he comentado en el apartado de herramientas, Unity permite crear figuras sencillas de manera predeterminada en la escena como planos, cubos, esferas...

Sin embargo para poder utilizar los modelos creados será necesario importarlos. El método más sencillo consiste en arrastrarlos directamente desde el directorio en el que se encuentran a la aplicación, aunque también pueden importarse desde una opción del menú.

Antes de comenzar a desarrollar la escena será necesario comentar algunos aspectos sobre el funcionamiento de Unity para facilitar la comprensión del proyecto y su desarrollo. El ciclo de vida de la ejecución de un proyecto en Unity consta de los siguientes pasos: Inicialización, Editor, Físicas, Eventos de Inputs, Lógica del Juego, Renderizado, GUI Rendering, Fin del frame, Pausa y Finalización. Durante la ejecución de un proyecto se inicializan en primer lugar todas las variables y se llama al script a ejecutar. A partir de aquí, empieza un bucle que abarcará desde la carga de las físicas a la pausa y, cuando este finaliza, se llama a la misma función en todos los objetos presentes en la escena para indicar el fin de la ejecución y posteriormente se deshabilita la escena. Durante el bucle central de la ejecución siempre se ejecutan en orden las siguientes acciones: se habilitan y calculan todas las físicas presentes en la escena como las animaciones, la gravedad o las colisiones; posteriormente se registran los inputs recibidos, se sigue la lógica registrada en el script llamando a los métodos update de los objetos que componen la escena y, por último, se renderiza la escena. Si la ejecución no se pausa, la ejecución del bucle continúa o se termina la ejecución del programa.

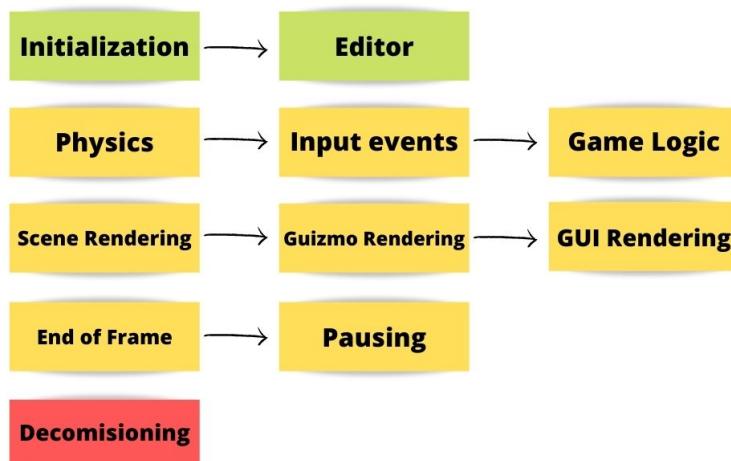


Figura 5.9: Orden de ejecución en Unity

Una vez comentado esto, comienzo con el desarrollo de la escena. Antes de colocar los modelos en la escena creo un plano que servirá como suelo en el que apoyarlos. Además, creo un material básico de color verde para asignárselo al suelo. Coloco ahora los modelos generados en el apartado anterior en la escena. Para poder colocar los modelos en la escena es necesario exportarlos junto con sus materiales y animaciones (en el caso del enemigo). Como el objetivo principal del proyecto es conseguir reducir la salud de la fortaleza lo máximo posible, coloco ésta en el centro y, a izquierda y

derecha de esta, coloco el almacén de oro y la torreta respectivamente. El enemigo lo coloco justo en frente de la fortaleza, de espaldas a la cámara (la cual elevo levemente para facilitar la visión de toda la escena), para poder observar con facilidad sus movimientos. Obtenemos así la siguiente escena, que a partir de ahora utilizaré para testear el algoritmo de Aprendizaje por Refuerzo.

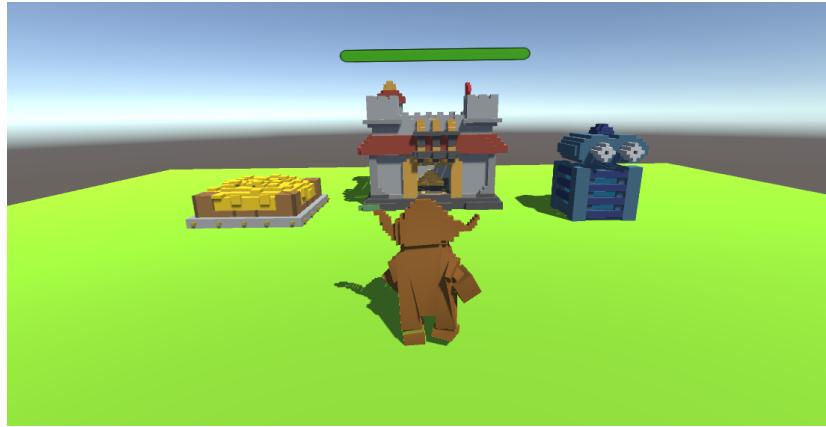


Figura 5.10: Escenario básico para el desarrollo del proyecto

Como se observa en la imagen, he incluido una barra que indica la salud de la fortaleza para visualizar más fácilmente el estado en el que se encuentra durante la ejecución del algoritmo. La implementación de esta se desarrollará en el siguiente apartado.

### 5.2.3. Implementación del algoritmo

A excepción del script que utilizaré para desarrollar el algoritmo de Aprendizaje por Refuerzo, todos los demás contendrán una clase que heredará de la clase *MonoBehaviour* de Unity, que es la clase base de la que heredan todos los scripts de Unity y que incluye las funciones básicas del ciclo de vida de la ejecución de un algoritmo comentada en el apartado anterior (ver fig 5.9). De manera predeterminada, al crear un *C# Script*, Unity genera un elemento que hereda de esta clase y crea automáticamente los métodos *Start()*, que se llama siempre al inicio de la ejecución del script y que usaré para inicializar las variables y *Update()*, que se llama en cada frame y utilizaré para actualizar los valores de algunas variables como la salud del enemigo, la fortaleza o el almacén de oro.

Antes de comenzar a implementar el algoritmo de Aprendizaje por Refuerzo, crearé un script para poder visualizar la salud de la fortaleza mediante una barra de vida. Para ello, añadimos un canvas a la fortaleza con un slider. El slider tendrá el fondo rojo y un relleno verde, de este modo, cuando la salud del enemigo se reduzca, la barra verde descenderá dejando visible el fondo rojo. A continuación muestro el código de las funciones perteneciente al script de la barra de salud:

```
void Start()
{
    health = maxHealth;
```

```

        slider.value = calculateHealth();
    }

    // Update is called once per frame
void Update()
{
    slider.value = calculateHealth();

    if (health <= 0) //Si la salud se reduce a cero
        destruimos el objeto asociado (Fortaleza)
    {
        Destroy(gameObject);
    }

}

//Funcion para calcular la salud actual
//Devuelve un valor entre 0 y 1 (que son los valores que
//toma el slider)
float calculateHealth()
{
    return health / maxHealth;
}

```

Extracto de código 5.1: FortalezaHealth.cs

En la función *Start()* inicializo la salud de la fortaleza al máximo y el valor del slider. Para calcular el valor del slider en cada momento se emplea la función *calculateHealth()*, que calcula un valor entre 0 y 1 (necesario para establecer el valor del slider correctamente), dividiendo su salud actual entre la salud máxima de la fortaleza. En la función *Update()* se actualizará el valor del slider cuando la fortaleza reciba daño y, en el caso de que la salud de la fortaleza llegue a 0, se destruye el objeto.

A continuación, desarollo el algoritmo de Aprendizaje por Refuerzo que utilizará la IA para el estudio de este trabajo. Para ello, utilizaré *ML-Agents*, un proyecto open-source que permite crear entornos para el entrenamiento de agentes inteligentes. Para poder utilizar correctamente *ML-Agents*, necesitaremos una versión de python igual o superior a la recomendada en la documentación. En este caso ya tengo instalada una versión de python superior a la recomendada (la versión 3.10.0). Ahora es posible crear un entorno virtual en el directorio donde se encuentra el proyecto de Unity mediante el comando:

```
python -m venv venv
```

Extracto de código 5.2: Crear un entorno virtual con python

El último parámetro que recibe el comando será el nombre del directorio que se cree, en este caso he decidido llamarlo *venv* para referirme al entorno virtual (Virtual Environment). Una vez creado, accedo al directorio *venv/Scripts* que acaba de crearse y

ejecuto *activate* desde la consola de comandos. Para poder continuar, es necesario comprobar que está instalada la versión mas reciente de pip. Una vez instalado, ejecutamos el primer comando para instalar el paquete *pytorch*:

```
pip install torch=1.11.0 -f https://download.pytorch.org/
    whl/torch_stable.html
pip install mlagents
mlagents-learn --help
```

Extracto de código 5.3: Instalación del paquete pytorch y mlagents

Dependiendo de la versión de *ML-Agents* que se desee utilizar será necesario comprobar qué versión de pytorch se necesita en la documentación del github. Mediante el segundo comando del extracto de código superior instalo el paquete mlagents y compruebo que se ha instalado correctamente ejecutando el último comando de ese extracto. Para finalizar los preparativos será necesario instalar ML Agents desde el gestor de paquetes de Unity. Una vez comprobado, comienzo a desarrollar el algoritmo en el proyecto de Unity.

#### 5.2.4. Prueba inicial

En primer lugar desarrollaré un algoritmo básico que permita a la IA avanzar hasta la fortaleza. Para ello, colocaré muros alrededor del escenario para que reciba una penalización si colisiona con ellos y recibirá un refuerzo positivo si colisiona con la fortaleza. El escenario resultante es el siguiente:

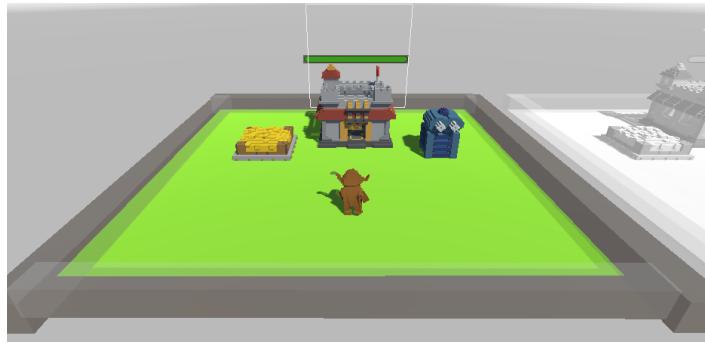


Figura 5.11: Escenario final para el desarrollo del proyecto

El primer paso será crear un script C# que llamaré *GoToGoal*, cuya clase heredará de la clase Agent. Para el correcto funcionamiento del algoritmo de aprendizaje por refuerzo será necesario que el agente realice una observación, tome una decisión, ejecute una acción y reciba un refuerzo (positivo o negativo) cíclicamente. De la observación se encargará la función *CollectObservations()*, que añade a las observaciones tanto la posición local del agente como la del objetivo (a continuación explicaré por qué trabajo con posiciones locales). La función *OnActionReceived()* se encargará de tomar una decisión y ejecutarla. Para ello, a partir de los valores continuos generados por el agente, la función se encargará de desplazarlo por los ejes X y Z a una velocidad determinada. Por último, la función *OnTriggerEnter()* se encargará de añadir una

recompensa positiva (si el agente llega a la fortaleza) o negativa (si el agente colisiona con un muro) y termina el episodio. A continuación se muestra parte de este código inicial:

```
public override void CollectObservations(VectorSensor  
    sensor)  
{  
    sensor.AddObservation(transform.localPosition);  
    sensor.AddObservation(targetTransform.localPosition);  
}  
  
public override void OnActionReceived(ActionBuffers  
    actions)  
{  
    float moveX = actions.ContinuousActions[0];  
    float moveZ = actions.ContinuousActions[1];  
  
    float moveSpeed = 10f;  
    transform.localPosition += new Vector3(moveX, 0,  
        moveZ) * Time.deltaTime * moveSpeed;  
}  
  
public override void Heuristic(in ActionBuffers  
    actionsOut)  
{  
    ActionSegment<float> continuousActions = actionsOut.  
        ContinuousActions;  
    continuousActions[0] = Input.GetAxisRaw("Horizontal");  
    ;  
    continuousActions[1] = Input.GetAxisRaw("Vertical");  
}  
  
private void OnTriggerEnter(Collider other)  
{  
    if(other.TryGetComponent<Fortaleza>(out Fortaleza  
        fortaleza))  
    {  
        Debug.Log(1);  
  
        AddReward(+1f);  
        EndEpisode();  
    }  
    if (other.TryGetComponent<Muro>(out Muro muro))  
    {  
        Debug.Log(-1);  
  
        SetReward(-1f);  
    }  
}
```

```

        EndEpisode();
    }
}

```

Extracto de código 5.4: Versión inicial del algoritmo

He empleado la función *Heuristic()* para poder comprobar funcionalidades manualmente sin necesidad de ejecutar el algoritmo de aprendizaje por refuerzo. Para acelerar el proceso de aprendizaje de la IA, clonaré el escenario de modo que se ejecuten varios escenarios simultáneamente (por esto es necesario trabajar con coordenadas locales en lugar de globales, ya que con coordenadas globales todos los personajes aparecerían en el mismo punto). Además, establezco un número máximo de pasos para la ejecución de modo que el agente no aprenda únicamente a esquivar los muros manteniéndose en una posición constante.

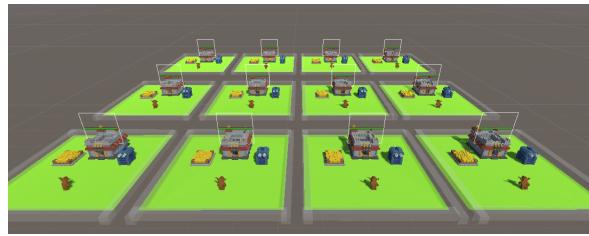


Figura 5.12: Entrenamiento de la IA en múltiples escenarios

Una vez que la IA consiga llegar hasta la fortaleza, el siguiente problema a resolver será que la IA haga daño a la fortaleza para intentar destruirla. Para ello será necesario añadir las siguientes variables y modificar algunas funciones del archivo *GoToGoal.cs*:

```

public FortalezaHealth saludFortaleza;
public float danoAFortaleza = 0f;

public override void OnEpisodeBegin()
{
    transform.localPosition = new Vector3(-0.8f, 5.9f, -15f
    );

    saludFortaleza.health = saludFortaleza.maxHealth;
}

public override void OnActionReceived(ActionBuffers
actions)
{
    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];
    danoAFortaleza = actions.ContinuousActions[2];

    float moveSpeed = 10f;

    transform.localPosition += new Vector3(moveX, 0,
    moveZ) * Time.deltaTime * moveSpeed;
}

```

```

}

private void OnTriggerEnter(Collider other)
{
    if(other.TryGetComponent<Fortaleza>(out Fortaleza
        fortaleza))
    {
        AddReward(+1f);

        //Si ha llegado hasta el objetivo ataca
        saludFortaleza.health -= Mathf.Abs(danoAFortaleza
            );
        AddReward(+Mathf.Abs(danoAFortaleza));

        if (saludFortaleza.health <= 0)
        {
            EndEpisode();
        }
    }
    if (other.TryGetComponent<Muro>(out Muro muro)) //Si
        se choca con un muro acaba el episodio
    {
        SetReward(-1f);
        EndEpisode();
    }
}

```

Extracto de código 5.5: Segunda versión del algoritmo para atacar la fortaleza

En primer lugar añado un objeto de la clase *FortalezaHealth* para poder acceder desde el enemigo a la salud de la fortaleza y creo otra variable (*danoAFortaleza*) con el daño que hará a la fortaleza en cada ataque, que será aleatorio al igual que el movimiento. Para asignar la fortaleza creada en el escenario a la variable *FortalezaHealth* creada simplemente basta con arrastrarla desde el editor de Unity al espacio generado para esta:

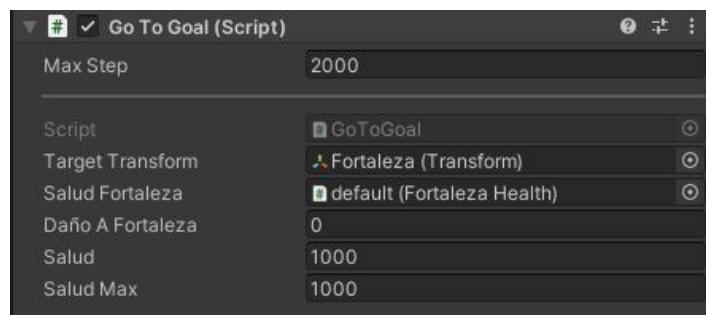


Figura 5.13: Variables en el editor de Unity

Para generar el daño que hace a la fortaleza en cada turno modifíco el tamaño

del vector `ContinuousActions` a 3 y añado en la función `OnActionReceived` una nueva línea para obtener el valor de `ContinuousActions`. En la función `OnTriggerEnter` añado ahora una nueva recompensa que dependerá del daño que haga a la fortaleza y reduzco la salud de esta. Cuando la salud de la fortaleza se reduzca a 0 el episodio terminará. Al inicio de cada episodio vuelvo a inicializar su salud en la función `OnEpisodeBegin`.

Una vez más compruebo su correcto funcionamiento mediante el empleo de la función `Heuristic` sin necesidad de ejecutar el entrenamiento. Para continuar con el desarrollo del algoritmo, haré que la torreta apunte hacia el enemigo constantemente y este reciba daño en función de la distancia a la que se encuentre de esta. Para ello, será necesario añadir una variable con la salud del enemigo y crear un script sencillo para la torreta. En `GoToGoal` será necesario añadir los siguientes fragmentos de código:

```
public float salud;
public float saludMax;

public override void OnEpisodeBegin()
{
    transform.localPosition = new Vector3(-0.8f, 5.9f, -15f
);

    //Volvemos a inicializar la salud del enemigo y la
    //fortaleza
    salud = saludMax;
    saludFortaleza.health = saludFortaleza.maxHealth;
}

private void Update()
{
    if(salud < saludMax){
        AddReward(salud - saludMax);
    }

    if (salud <= 0)
    {
        EndEpisode();
    }
}
```

Extracto de código 5.6: Añadidos a `GoToGoal` para indicar la salud el enemigo

Al igual que en la fortaleza añado dos variables nuevas, una para la salud máxima del enemigo y otra para su salud actual. Del mismo modo, cada vez que comienza un nuevo episodio se reinicia la salud del enemigo al máximo en la función `OnEpisodeBegin`. Creo además la función `Update()`, que comprobará en cada llamada si la salud del enemigo se ha reducido, añadiendo en ese caso un refuerzo negativo que será la diferencia de su salud actual con el máximo. Además, si la salud de este se reduce a 0, el episodio termina.

A continuación muestro el script que creo para la torreta:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Torreta : MonoBehaviour
{
    public GoToGoal enemigo;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        //Se reduce la salud del enemigo segun la funcion
        gaussiana
        if (enemigo.salud > 0)
        {
            float x = Mathf.Pow((enemigo.transform.
                localPosition.x - transform.localPosition.
                x),2)/2;
            float y = Mathf.Pow((enemigo.transform.
                localPosition.y - transform.localPosition.
                y),2)/2;
            enemigo.salud-= Mathf.Exp(-(x+y));

            transform.LookAt(enemigo.transform);
        }
    }
}

```

Extracto de código 5.7: Algoritmo de la torreta

Como comenté al principio, este algoritmo se asocia a la parte superior de la torreta para que ésta pueda girar mientras la parte inferior se mantiene estática. Al igual que en el enemigo, creo una variable del tipo *GoToGoal*, que contiene la información sobre la salud de éste, y se la asocio desde el editor de Unity. La principal función es la función *Update()*, que en cada frame actualiza el ángulo al que mira la torreta mediante la función *LookAt(enemigo.transform)* que recibe como parámetro la localización del objetivo. Además, esta función comprueba si la salud del enemigo es positiva y, en ese caso, la reduce según una función gaussiana de dos dimensiones. De este modo el enemigo recibirá más daño cuanto más cerca se encuentre de la torreta y menos daño

cuanto más se aleje.

$$f(x, y) = Ae^{-\left(\frac{(x-x_0)^2}{2} + \frac{(y-y_0)^2}{2}\right)} \quad (5.1)$$

En este caso A vale 1,  $x_0$  e  $y_0$  representan las coordenadas de la torreta, y x e y son las coordenadas del enemigo en cada frame. Cuando el objetivo esté cerca de la torreta el daño que recibirá será cada vez más cercano a 1 y cuando se aleje este tenderá a 0.

## 5.3. Pruebas

### 5.3.1. Introducción

En este capítulo explicaremos...

### 5.3.2. Conclusiones

En este capítulo concluimos que...

---

## **6. Conclusiones y vías futuras**

---

---

## 7. Bibliografía

---

- [1] Adobe. Página principal de mixamo, 2022. URL <https://www.mixamo.com/#/>.
- [2] Alfa Beta. Noticia con información sobre la ia del videojuego hello neighbor 2, 2021. URL <https://alfabetajuega.com/xbox/hello-neighbor-2-ofrece-nuevos-detalles-de-su-ia>.
- [3] MagicaVoxel. Página principal de magica voxel, 2021. URL <https://ephtracy.github.io/>.
- [4] Unity Technologies. Página principal de unity, 2020. URL <https://unity.com/es>.
- [5] Unity Technologies. Repositorio en github de ml-agents, 2022. URL <https://github.com/Unity-Technologies/ml-agents>.
- [6] Unity Technologies. Ml-agents en la página oficial de unity, 2022. URL <https://unity.com/es/products/machine-learning-agents>.