



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

1. Resumen	1
2. Abstract	2
3. Introducción	3
3.1. Contexto del proyecto	3
3.2. Problema abordado	4
3.3. Técnicas y conceptos empleados	4
3.3.1. Aprendizaje por Refuerzo	5
3.3.2. Modelado Voxel Art	7
3.3.3. Creación de animaciones	8
3.3.4. Creación del videojuego	9
3.4. Contenido de la memoria	11
4. Objetivos	12
5. Desarrollo del trabajo	13
5.1. Análisis y diseño	13
5.1.1. Planificación y presupuesto	13
5.1.2. Requerimientos	15
5.1.3. Requisitos funcionales	15
5.1.4. Requisitos funcionales	16
5.1.5. Diseño gráfico	17
5.2. Implementación	19
5.2.1. Diseño de la escena	19
5.2.2. Implementación del algoritmo	21
5.3. Pruebas	31
5.3.1. Evolución del entrenamiento durante el proceso de desarrollo	32
5.3.2. Valores de las recompensas	35
5.3.3. Análisis de los resultados obtenidos	35
5.3.4. Aplicación de las IAs obtenidas	38
6. Conclusiones y vías futuras	42
7. Bibliografía	43

Índice de figuras

3.1. Esquema del funcionamiento de un algoritmo de Aprendizaje por Refuerzo	5
3.2. Herramientas de Magica Voxel	8
3.3. Ejemplo de material cristal, metal y nuboso	8
3.4. Orden de ejecución en Unity	10
3.5. Herramientas empleadas	11
5.1. Planificación real del primer cuatrimestre	13
5.2. Planificación real del segundo cuatrimestre	14
5.3. Ciclo de vida del enemigo	16
5.4. Boceto del escenario de la aplicación	16
5.5. Boceto y modelo del enemigo básico	17
5.6. Enemigo básico: Modelo renderizado	18
5.7. Definición del esqueleto del modelo	18
5.8. Estructuras: Modelos renderizados	19
5.9. Tipos de proyecto disponibles en Unity	20
5.10. Escenario básico para el desarrollo del proyecto	21
5.11. Diagrama de clases	22
5.12. Escenario final para el desarrollo del proyecto	24
5.13. Entrenamiento de la IA en múltiples escenarios	25
5.14. Variables en el editor de Unity	27
5.15. Leyenda para los entrenamientos realizados durante el desarrollo	32
5.16. Cumulative Reward y Episode Length	32
5.17. Policy Loss y Value Loss	33
5.18. Beta y Entropy	34
5.19. Extrinsic Reward, Extrinsic Value Estimate y Learning Rate	34
5.20. Leyenda para los entrenamientos finales	36
5.21. Cumulative Reward y Episode Length (Prueba final)	36
5.22. Policy Loss y Value Loss (Prueba final)	37
5.23. Beta y Entropy (Prueba final)	37
5.24. Extrinsic Reward, Extrinsic Value Estimate y Learning Rate (Prueba final)	37
5.25. Asignar la IA al enemigo creado	38
5.26. Gráfico Q-Q Plot	41

Índice de extractos de código

5.1.	FortalezaHealth.cs	22
5.2.	Crear un entorno virtual con python	23
5.3.	Instalación del paquete pytorch y mlagents	23
5.4.	Versión inicial del algoritmo	24
5.5.	Segunda versión del algoritmo para atacar la fortaleza	25
5.6.	Añadidos a GoToGoal para indicar la salud el enemigo	27
5.7.	Torreta.cs	28
5.8.	Añadidos al agente para interaccionar con el almacén de oro	29

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, impulsada también por la popularización del uso de unidades de procesamiento gráfico (GPUs) en ordenadores, dispositivos móviles y consolas, 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 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, driven by the popularisation of the use of graphics processing units (GPUs) in computers, mobile devices and consoles, 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 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. En el libro de Georgios N. Yannakakis y Julian Togelius [9] podemos encontrar ejemplos sobre cómo se podrían implementar una IA para un *Pacman*, usando distintos métodos de Aprendizaje Automático, así como las ventajas y desventajas de cada uno de estos métodos y la eficiencia de cada uno en distintos tipos de videojuegos. 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 usando redes neuronales que permitan al enemigo observar y aprender de las acciones del jugador para actuar según el método de juego de cada uno. De este modo, cada partida jugada contribuye al desarrollo y el entrenamiento de la IA del enemigo.

Existen estudios similares, como el estudio de rendimiento Minimax y agentes de Aprendizaje por Refuerzo en el juego *iwoki* [8], que han observado el comportamiento de distintos agentes jugando al juego *iwoki math*. En él se realiza un estudio sobre qué implementaciones son más adecuadas para agentes que pretenden jugar a un juego de mesa evaluando distintas configuraciones de Aprendizaje por Refuerzo y comprobando qué configuraciones optimizan los resultados. Por tanto, mi proyecto ayudará a complementar otros similares que analizan, del mismo modo, distintas configuraciones de entrenamientos de Aprendizaje por Refuerzo para comprobar cuáles aportan

mejores resultados en un campo concreto. Así mismo, existen estudios que analizan algunos casos de Aprendizaje por Refuerzo en videojuegos actuales, como el estudio sobre Aprendizaje por Refuerzo en Videojuegos [4]. Sin embargo, a pesar de que se realiza un análisis bastante exhaustivo, estos estudios están más centrados en el aspecto teórico y no comprueban los resultados obtenidos por cada uno de los métodos que emplean. Por esto mismo, este trabajo proporcionará los distintos resultados obtenidos para cada entrenamiento, para así poder observar y comparar qué métodos son más eficaces a la hora de generar una IA.

3.2. Problema abordado

El objetivo de este trabajo consistirá en modelar distintos componentes de un videojuego mediante *VoxelArt* y desarrollar una IA mediante técnicas de Reinforcement Learning (Aprendizaje por Refuerzo) que aprenda del entorno que la rodea, pudiendo insertar luego el modelo entrenado del agente en un proyecto para comprobar su funcionamiento. De este modo, comprobaré si se pueden aplicar conocimientos de Aprendizaje por Refuerzo en videojuegos en un entorno dinámico no estudiado y estudiar los resultados obtenidos con entrenamientos para distintas recompensas. Así mismo, comprobaré las ventajas que aporta cada uno de estos entrenamientos y observaré si estos métodos podrían extenderse en la industria de manera factible para dar lugar a experiencias de juego más ricas.

El videojuego constará de una escena sencilla compuesta de cuatro elementos principales: un almacén de oro, una fortaleza, una torreta y un enemigo básico. El objetivo del proyecto será que el enemigo básico, que usará una IA entrenada por Aprendizaje por Refuerzo, cause la mayor cantidad de daño posible a la fortaleza que se encuentra en la escena. Para ello, el enemigo deberá ser consciente de su entorno y evitar la torreta, que intentará eliminarlo de la escena, evitar salirse del mapa en el que se encuentra y aprovechar el almacén de oro para intentar conseguir mejoras como un aumento de daño. Todos los detalles sobre las interacciones del enemigo con la escena pueden encontrarse en el apartado [5.1](#) en la sección de Requerimientos.

3.3. Técnicas y conceptos empleados

Antes de comenzar con el desarrollo del proyecto comentaré las técnicas y conceptos matemáticos e informáticos que emplearé para su elaboración. La principal fuente de información empleada para aprender sobre los fundamentos y los conceptos más importantes del Aprendizaje por Refuerzo que he empleado se trata del libro *Artificial Intelligence and Games* [9]. En la bibliografía se encuentra también un enlace a un artículo que profundiza en qué es el voxel art y algunos ejemplos de su uso [3].

3.3.1. Aprendizaje por Refuerzo

El Aprendizaje por Refuerzo es un enfoque del aprendizaje automático basado en la forma en la que los humanos y los animales aprenden a tomar decisiones basándose en un sistema de recompensas, que pueden ser tanto positivas como negativas, que reciben del entorno que les rodea. En el Aprendizaje por Refuerzo, en esencia, las señales que recibe el algoritmo para entrenarse se basan en cómo el agente interactúa con el entorno. Dado un momento t del tiempo, un agente se encuentra en un estado s y debe tomar una acción a entre todas las disponibles en su estado actual, recibiendo del entorno como respuesta una recompensa r . Así, el agente aprende de manera gradual mediante sus interacciones con el entorno a maximizar su recompensa.

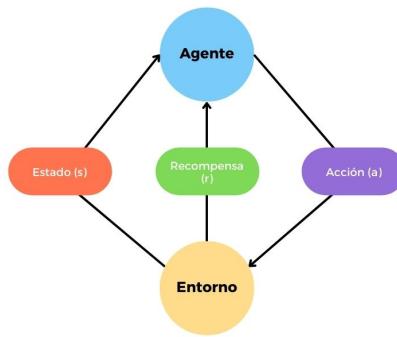


Figura 3.1: Esquema del funcionamiento de un algoritmo de Aprendizaje por Refuerzo

Descrito de una manera más formal, el objetivo del agente será descubrir una política (π) de selección de acciones que maximice la recompensa obtenida a largo plazo, siendo esta política una estrategia de selección de acciones dado el estado en el que se encuentra el agente. En el caso de que exista o se aprenda la función que caracteriza el valor de cada acción, la política óptima puede obtenerse seleccionando la acción con el valor más alto. Cada interacción con el entorno ocurrirá en momentos discretos del tiempo y se modela como un proceso de decisión de Markov (MDP). El MDP se define por: S , un conjunto de estados con la información del agente sobre el entorno; A , un conjunto de acciones posibles en cada estado en las que el agente interactúa con el entorno; $P(s,s',a)$, una función que calcula la probabilidad de transición del estado s a s' después de tomar una decisión a y $R(s,s',a)$, la función que devuleve una recompensa por pasar del estado s a s' dada una acción a . Hay que destacar que la función $P(s,s',a)$ cumple la propiedad de Markov, es decir, que los futuros estados del proceso dependen únicamente del estado en el que se encuentra actualmente y no la secuencia de acciones que le preceden. Las funciones P y R definen el modelo del mundo y representan las dinámicas del entorno y las recompensas a largo plazo respectivamente.

Uno de los problemas centrales del Aprendizaje por Refuerzo reside en encontrar el equilibrio adecuado entre explotar los conocimientos aprendidos actuales del algoritmo y explorar nuevos territorios del espacio de búsqueda. Tanto seleccionar aleatoriamente acciones sin tener en cuenta los conocimientos aprendidos como elegir siempre

la mejor acción sin explorar son estrategias que suelen dar malos resultados. Uno de los enfoques más populares y eficientes para la selección de acciones es el ϵ -greedy, determinado por el parámetro $\epsilon \in [0, 1]$. En este, el agente elige la acción que cree que devolverá la recompensa más alta con probabilidad $1 - \epsilon$ y, en otro caso, elige una acción aleatoria.

Otros dos conceptos fundamentales en el Aprendizaje por Refuerzo son los conceptos de *Bootstrapping* y *backup*. El Bootstrapping es una noción que estima lo bueno que es un estado basándose en lo bueno que cree que es el siguiente estado, es decir, actualiza una estimación de un estado basándose en otra estimación. Por otro lado, la noción de backup consiste en ir hacia atrás desde un estado en el futuro s_{t+h} al estado actual que queremos evaluar, s_t , y considera los valores intermedios entre ambos estados. Sus dos propiedades principales son su profundidad, que puede variar desde un paso hacia atrás a una revisión completa; y su amplitud, que también puede variar desde un número aleatorio de estados en cada paso del tiempo a toda una revisión completa en amplitud. Para el algoritmo que desarrollaré la profundidad del backup será de 1 para agilizar el entrenamiento de la IA, ya que se entrenará simultáneamente en diversos escenarios.

De acuerdo al capítulo 2.6.1 del libro [9], los problemas de Aprendizaje por Refuerzo pueden clasificarse de distintas formas. Según la clasificación ahí realizada, mi proyecto se clasificaría como **episódico**, en el que el algoritmo se produce dentro de un horizonte finito de múltiples instancias de entrenamiento. Se denomina **episodio** a la secuencia finita de estados, acciones y recompensas recibidas dentro de ese horizonte. Otro ejemplo de RL (Reinforcement Learning) episódico serían los métodos Monte Carlo que se basan en el muestreo aleatorio repetido. Según la segunda clasificación realizada en este apartado, el algoritmo que desarrollaré se tratará también de un algoritmo **off-policy**, en el que la política será independiente de las acciones del agente, es decir, el agente recibirá recompensas en función de cada acción que realice y no según toda la secuencia de acciones previas al estado en el que se encuentra. Para exemplificar esto, el enemigo que contendrá el algoritmo de Aprendizaje por Refuerzo recibirá una recompensa cuando consiga atacar la fortaleza, sin embargo, esta recompensa no dependerá de las acciones anteriores que haya tomado, por lo que si el agente realiza un trayecto más largo para llegar a la fortaleza (por ejemplo), su recompensa al atacarla será la misma que si llegase por un camino más corto.

Un algoritmo muy usado en Aprendizaje por Refuerzo se trata del algoritmo *Q-learning*. Aunque este algoritmo no es posible implementarlo en ML-Agents, que es la herramienta de Unity que utilizaré para crear el algoritmo de mi enemigo, comentaré cómo funciona para obtener una visión más clara de algunos algoritmos de Aprendizaje por Refuerzo. Este método utiliza una función $Q(s, a)$, que representa la recompensa por tomar una acción a en el estado s . De este modo el algoritmo contiene una tabla que se actualiza después de ejecutar una acción con el valor de la recompensa obtenida. Inicialmente, la tabla contiene valores aleatorios establecidos por el diseñador del problema. Después, cuando el agente toma una acción a en un estado s , recibe una recompensa y la tabla se actualiza siguiendo la siguiente fórmula:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max Q(s', a') - Q(s, a)] \quad (3.1)$$

En esta fórmula, $\alpha \in [0, 1]$ es la tasa de aprendizaje (Learning Rate) y $\gamma \in [0, 1]$ es el factor de descuento (Discount Factor). La tasa de aprendizaje determinará hasta que punto la nueva estimación de Q anulará la estimación anterior. El factor de descuento, por otra parte, pondera la importancia de las recompensas anteriores frente a las posteriores, es decir, cuanto más cerca esté γ de 1, más importancia se le darán a los refuerzos futuros. Este algoritmo usa bootstrapping, ya que mantiene estimaciones de lo bueno que es un par estado-acción en función de lo bueno que cree que será el siguiente par. También utiliza una copia de seguridad (backup) de un paso de profundidad para estimar el nuevo valor de Q teniendo en cuenta los valores Q de las posibles acciones a' del nuevo estado visitado s' .

A pesar de esto, el método tiene varias desventajas. En primer lugar, el tamaño del espacio estado-acción puede ser muy costoso de manejar computacionalmente. Además, a medida que aumenta el tamaño de la tabla Q , se incrementan las necesidades computacionales para asignar memoria e información. Por último, se podría experimentar una convergencia demasiado larga ya que el tiempo de aprendizaje es exponencial al tamaño del espacio estado-acción.

3.3.2. Modelado Voxel Art

Uno de los estilos artísticos más famosos en la actualidad, principalmente debido a la nostalgia que transmite, es el pixel art, que he comentado en el apartado anterior. Un voxel es el equivalente a un píxel en forma volumétrica o en 3D. Surge así a partir de este estilo el voxel art, que consiste en utilizar métodos similares a los empleados para el pixel art pero en tres dimensiones. Esto permite generar modelos y entornos con un apartado artístico similar al que obtendríamos si usáramos pixel art pero en 3D. Aunque su uso surgió en la década de los 90 no ha sido hasta la última década que se ha popularizado su uso. La principales ventajas de este estilo es que permite representar casi cualquier sólido de manera aproximada. Para el modelado de este proyecto con este estilo artístico utilizaré el programa Magica Voxel, 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 sencillamente 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.

Otra de las ventajas de trabajar con voxels reside en la facilidad para implementar operaciones booleanas. Magica Voxel es un programa de modelado que permite trabajar fácilmente con los voxels aprovechándose de estas ventajas que he comentado. Para ello, utiliza una rejilla segmentada en voxels y emplea herramientas que permiten vaciar, añadir, extruir... los voxels que la forman. A continuación muestro una captura del panel de herramientas de Magica Voxel.

Como se observa en la captura el programa incluye herramientas para realizar

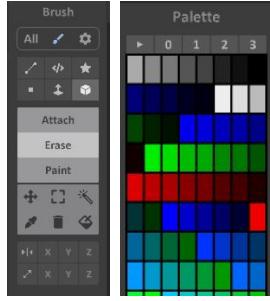


Figura 3.2: Herramientas de Magica Voxel

operaciones de extrusión, barrido, escalado, etc. Sin embargo, no sólo se permite realizar modelos. El programa también permite asociar una textura a cada voxel mediante una paleta personalizable. El programa asociará a cada color unas propiedades pre-determinadas que se pueden modificar en el panel de renderizado. Estas propiedades permiten que todos los voxels que tengan asociados ese color cumplan distintas propiedades como reflejar la luz, dejar pasar la luz, etc. Entre las propiedades que ofrece el programa se encuentran las posibilidades de crear: materiales difusos (valor por defecto); materiales emisivos, que actúan como una fuente de luz pudiendo modificarse la cantidad de luz que se emite y la intensidad de esta; materiales metálicos, modificando la rugosidad del material y el índice refractivo (que indica la cantidad de luz que refleja el objeto); materiales de cristal, que permiten modificar los mismos valores que los metálicos y además modificar la transparencia del material y, por último, materiales nubosos, que genera un efecto similar a una niebla y cuyo único parámetro modificable es la densidad de la niebla.

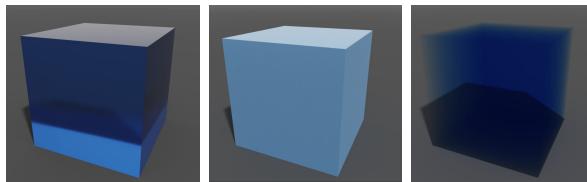


Figura 3.3: Ejemplo de material cristal, metal y nuboso

3.3.3. Creación de animaciones

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.

3.3.4. Creación del videojuego

Para diseñar la escena y poder implementar el videojuego junto con el algoritmo de Aprendizaje por Refuerzo he empleado **Unity** [5], 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.

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 fase de **Inicialización** se llaman a dos funciones en cada uno de los objetos que se encuentran en la escena. Estas dos funciones, las funciones *Awake* y *OnEnable*, se llamarán cuando se instancie a un objeto de la escena antes de cualquier función de inicio para activarlos. Posteriormente, en la fase **Editor**, se inicializarán las propiedades de los scripts cuando estos se adjuntan por primera vez a un objeto o se utilice el comando *Reset*, tras esto, se llamará a la función *Start* que iniciará la ejecución del script.

A partir de aquí, empieza un bucle que abarcará desde la fase de Físicas a la de Pausa y que se ejecutará por cada frame. La fase de **Físicas**, cuyo ciclo puede ocurrir más de una vez por frame si el tiempo de paso es menor que el tiempo de actualización del frame, se realizan todos los cálculos y actualizaciones de física presentes en la escena, entre ellos se encuentra el cálculo de colisiones. Además, en esta fase se llaman a todos los eventos de animación para los clips desde la última actualización hasta la llamada actual. La fase de **Eventos de Inputs**, se encarga de registrar los eventos de inputs realizados en la escena mediante el ratón o el dispositivo utilizado. La fase **Lógica del juego** contiene la llamada a la función *Update*, que será la principal función que utilizaré para actualizar cada frame. En esta fase se vuelve a realizar una actualización interna de las animaciones.

Durante las fases de **Renderizado**, se calcula qué objetos se encuentran dentro de la escena, si son visibles o no y se renderizan (**Scene Rendering**); si se está trabajando simultáneamente con el editor se dibujan todos los Guizmos en la escena, que pueden ser figuras geométricas, texturas, etc. (**Guizmo Rendering**) y, por último, se llama

múltiples veces por cada frame la función *OnGUI*, que se encarga de responder a todos los eventos de GUI llamando a los *Repaint* necesarios y a los eventos de teclado o ratón asociados a cada evento de entrada (**GUI Rendering**).

Por último, en la fase **Fin del frame** se llama a las funciones que necesitan esperar a que el frame haya acabado para su ejecución y en la fase de **Pausa** se detiene la ejecución temporalmente si se detecta que se ha indicado una pausa, aunque se ejecutará otro frame antes de la detención.

Al salir, en la fase de **Finalización**, se llama en todos los objetos activos en la escena a las funciones correspondientes para desactivarlos y realizar todos los cambios necesarios antes de cerrar la aplicación.

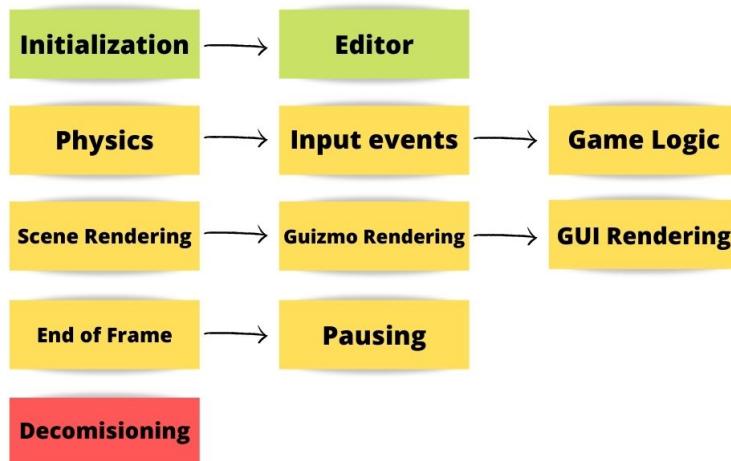


Figura 3.4: Orden de ejecución en Unity

Por último, para implementar el algoritmo, utilizaré **ML-Agents** [6]. 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 [7], 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.

ML-Agents permite realizar entrenamientos estructurados por episodios. Los episodios son un margen de tiempo en el que el agente deberá moverse e interactuar con

el entorno tomando distintas decisiones y recibiendo recompensas. Los episodios tienen una duración máxima delimitada por el programador que indicará el número máximo de pasos que puede tomar el agente antes de que termine el episodio. Sin embargo, un episodio puede terminar sin necesidad de alcanzar el número máximo de pasos (por ejemplo, si el agente colisiona con un muro o su salud se reduce a 0). Durante cada entrenamiento, se ejecutarán múltiples episodios y, tras ejecutar un gran número de estos, ML-Agents calcula la recompensa media obtenida. Todas estas recompensas se almacenan en un fichero que posteriormente se analizará para comprobar los resultados obtenidos. Además de la recompensa media, ML-Agents almacena otros valores como la pérdida de la política o la entropía, en los cuales profundizaré más en el apartado de Pruebas.



Figura 3.5: Herramientas empleadas

3.4. Contenido de la memoria

El proceso para el desarrollo de este proyecto se encuentra estructurado en esta memoria siguiendo la siguiente estructura por apartados:

La sección 4 contendrá los objetivos previstos para el desarrollo de este proyecto, así como los finalmente alcanzados. Además, conectaré cada uno de los objetivos con los distintos apartados de la memoria para facilitar y agilizar su comprensión.

La sección 5 contendrá una explicación de todos los métodos que he empleado para desarrollar el trabajo y conseguir cumplir con los objetivos previstos. Esta sección se dividirá a su vez en varios apartados. En primer lugar, mostraré el proceso de análisis y diseño del proyecto, que incluirá la planificación y el presupuesto necesarios para desarrollarlo, los requerimientos de la aplicación y el proceso de modelado. A continuación, mostraré todo el proceso para implementar la escena y el algoritmo de aprendizaje por refuerzo. Por último, contendrá un apartado con todas las pruebas realizadas para distintos entrenamientos y un análisis de los resultados.

La sección 6 contendrá una reflexión sobre los resultados obtenidos con el desarrollo del proyecto y posibles vías futuras para poder seguir ampliando este estudio y sus aplicaciones en la industria del videojuego.

Por último, se incluye una sección con la bibliografía consultada para realizar este proyecto.

4. Objetivos

En este capítulo comentaré los objetivos previstos en el desarrollo del proyecto, así como los finalmente alcanzados, indicando las posibles dificultades encontradas durante el proceso de desarrollo, los cambios y posibles mejoras realizadas en este.

El principal objetivo de este proyecto será crear un escenario con distintos elementos que permitan hacer simulaciones donde el enemigo principal, que contendrá una Inteligencia Artificial mejore su comportamiento mediante Aprendizaje por Refuerzo. Para poder cumplir este objetivo será necesario crear modelos para un avatar inteligente e implementar la IA que utilizará. Esta IA deberá poder entrenarse para, posteriormente, aplicarse a un avatar inteligente en un escenario y comprobar la eficiencia de esta. Asimismo, otro objetivo será necesario modelar estructuras para dar lugar a un escenario con el que el avatar pueda interactuar para entrenar su algoritmo de Aprendizaje por Refuerzo.

Para modelar todas las estructuras utilizaré el programa Magica Voxel mediante Voxel Art, técnica que he comentado en el apartado anterior. Una descripción más exacta de cómo interactuará el enemigo con las distintas estructuras se encuentra a continuación en el apartado *5.1.3 Requerimientos* del capítulo [5.1](#). El modelado, por otra parte, puede encontrarse detallado en el apartado *5.1.5 Modelado*, del mismo capítulo.

EL objetivo principal relacionado con el desarrollo del algoritmo de Aprendizaje por Refuerzo será implementar una IA que se pueda aplicar en un agente para un videojuego, de manera que este sepa interactuar con los objetos de su entorno independientemente de la localización de estos. Para ello, implementaré un algoritmo mediante el uso del paquete de Unity ML-Agents, que describo con más detalle en el apartado de herramientas del capítulo anterior. Tras esto, entrenaré esta IA en múltiples escenarios distintos para que aprenda a interactuar con los modelos descritos y, finalmente, aplicaré la IA entrenada al enemigo creado para comprobar la eficiencia de este método y su viabilidad en el desarrollo de videojuegos. Para poder observar correctamente el comportamiento de la IA y la eficiencia de los entrenamientos, entrenaré la IA modificando y probando distintos valores de recompensas para las acciones. Una vez obtenidos los resultados tras cada entrenamiento, los compararé para comprobar qué recompensas han sido más eficientes al entrenar el algoritmo y han dado lugar a comportamientos más eficientes.

La implementación de todo el algoritmo de Aprendizaje por Refuerzo se encuentra en el apartado [5.2](#), donde explicaré todo el procedimiento seguido para implementar el comportamiento, tanto del agente como de las estructuras que participarán en la escena. Los resultados obtenidos en los entrenamientos pueden observarse en el apartado [5.3](#), donde expondré los resultados obtenidos en los entrenamientos, los distintos valores aplicados a las recompensas y una comparativa entre los resultados obtenidos para descubrir qué método ha sido más efectivo a la hora de obtener una IA eficiente.

5. Desarrollo del trabajo

5.1. Análisis y diseño

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

5.1.1. Planificación y presupuesto

Para realizar una planificación general, dedicaré el primer 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: la primera semana consistirá en planificar el trabajo durante los próximos meses. Las siguientes tres semanas se emplearán para recabar y estudiar documentos (sobre técnicas de aprendizaje por refuerzo, modelado de personajes, etc.) para el desarrollo del proyecto. Esto podrá retrasarse como máximo hasta la quinta semana de desarrollo. Durante el período de la cuarta a la séptima semana deberá instalar y preparar todas las herramientas necesarias para el desarrollo del proyecto, así como comprobar el correcto funcionamiento de las aplicaciones y solucionar los posibles problemas. La octava semana d

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.

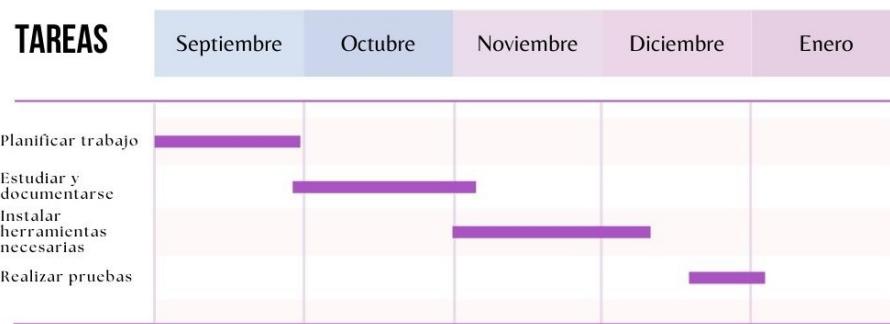


Figura 5.1: Planificación real del primer cuatrimestre

Planificación del segundo semestre: En la primera semana de este cuatrimestre empezaré a realizar bocetos del personaje principal que actuará en la escena y las estructuras que se encontrarán presentes en esta. Durante la segunda y tercera semana, modelaré a este personaje y estructuras. Las semanas cuarta y quinta crearé alguna animación para el personaje y la importaré a la plataforma de desarrollo del videojuego para comprobar que funciona correctamente. Desde la semana seis a la once, me dedicaré exclusivamente a desarrollar el videojuego y el algoritmo de Aprendizaje por Refuerzo, así como a empezar a desarrollar la memoria del trabajo. Para la semana número 12 necesitaré tener una versión básica funcional (al menos) del algoritmo de Aprendizaje por Refuerzo para poder revisar el progreso y desarrollo de este. La memoria se realizará durante todo este tiempo hasta la última semana de desarrollo (una semana y media antes de la entrega).

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 otros imprevistos que realientizaron el desarrollo, provocó un retraso de una semana aproximadamente en el desarrollo de las actividades previstas.

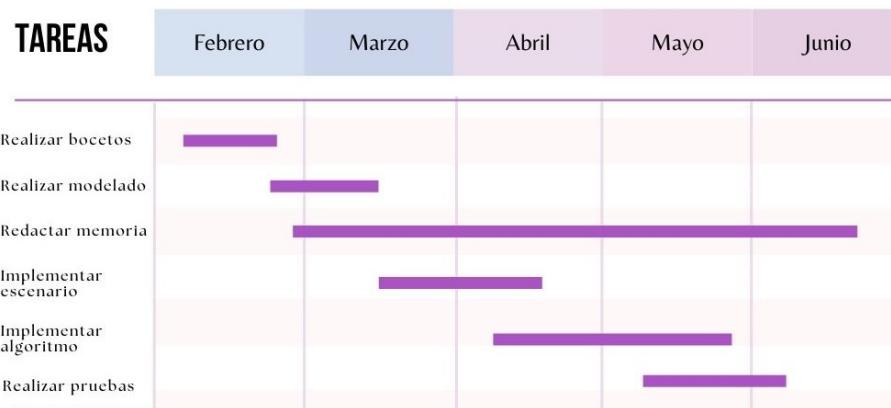


Figura 5.2: Planificación real del segundo cuatrimestre

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€.

A continuación realizo un análisis de los requisitos funcionales y no funcionales

	Precio
Ordenador MSI	900€
Programador	20€/hora
Total	9900€

Cuadro 5.1: Presupuesto del trabajo

del proyecto, así como de otros requerimientos generales necesarios para completar con éxito el desarrollo de este.

5.1.2. Requerimientos

Las estructuras que modelaré para el desarrollo del proyecto serán las siguientes: Una **fortaleza** con una gran cantidad de salud, siendo esta la estructura más grande de las tres que modelaré. El objetivo del agente será reducir en la mayor cantidad posible la salud de la fortaleza. Un **almacén de oro**, que será de menor tamaño que la fortaleza y contendrá una cantidad determinada de oro. Esta estructura permitirá al enemigo obtener alguna bonificación al obtener oro de esta. Una **torreta**, cuyo modelado separaré en una parte superior y otra inferior permitiendo, de este modo, que la parte superior de la torreta gire mientras la parte inferior se mantiene estática. Esta estructura apuntará en todo momento al enemigo infligiéndole daño, de modo que el algoritmo de Aprendizaje por Refuerzo deberá aprender a evitar esta estructura siempre que sea posible. El **enemigo**, será un modelo que se encontrará haciendo una *pose en T*, lo que me permitirá crearle animaciones definiendo una serie de puntos en el modelo que formarán sus articulaciones.

5.1.3. 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. Los ataques del enemigo consistirán en golpes con el puño a las estructuras, por lo que deberá colisionar con ellas para atacarlas. 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 5.3.

El proyecto constará de un sólo nivel, aunque las posiciones en las que se generen

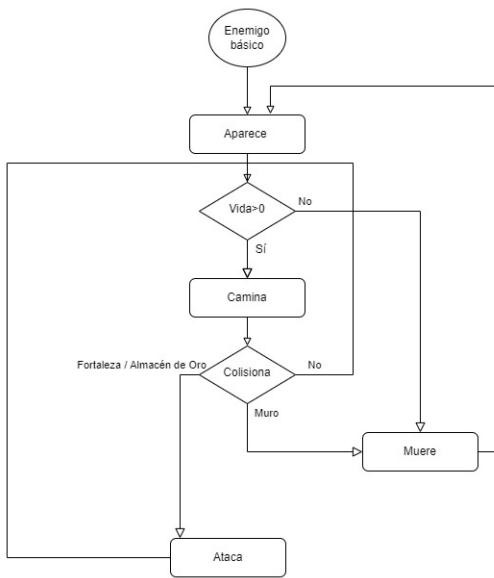


Figura 5.3: Ciclo de vida del enemigo

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:

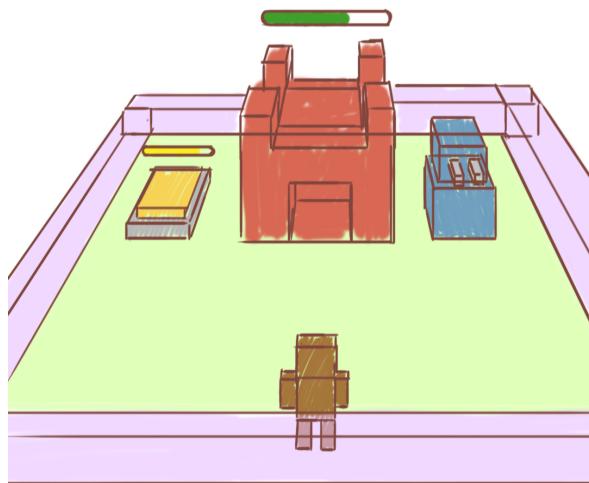


Figura 5.4: Boceto del escenario de la aplicación

5.1.4. Requisitos funcionales

Requisitos no funcionales: la aplicación será una apliación ejecutable en PC desde el escritorio. El framerate de la aplicación se mantendrá estable y nunca deberá ser inferior a los 30 FPS (Frames Por Segundo).

5.1.5. Diseño gráfico

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*. Además, como comenté anteriormente, es necesario realizar el modelado del enemigo realizando una pose en T, ya que será un requisito fundamental para poder definir las animaciones posteriormente en *Mixamo*. El modelo del personaje obtenido puede observarse en las figuras 5.5 y 5.6.

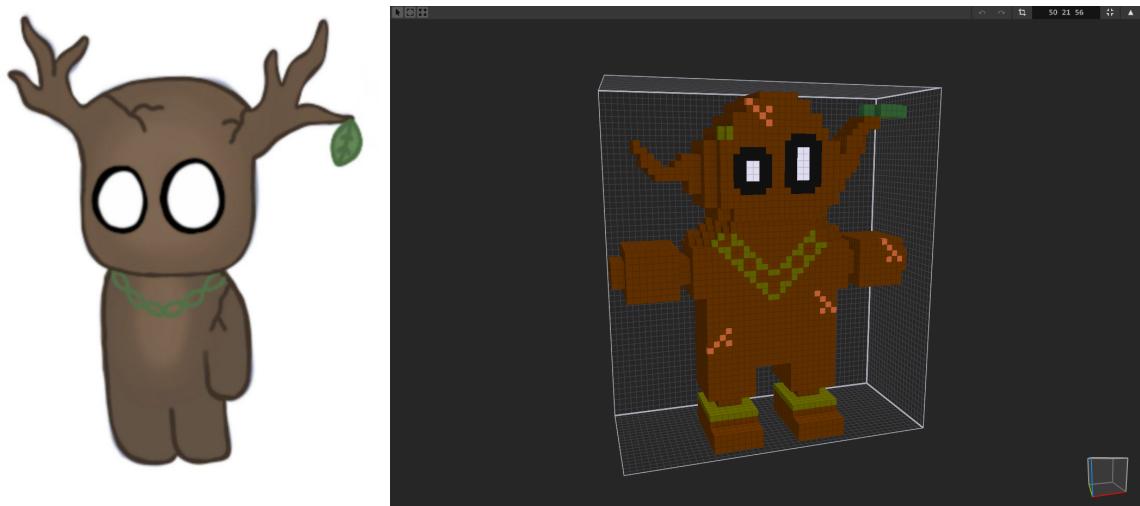


Figura 5.5: Boceto y modelo del enemigo básico

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 en la figura 5.7 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,



Figura 5.6: Enemigo básico: Modelo renderizado

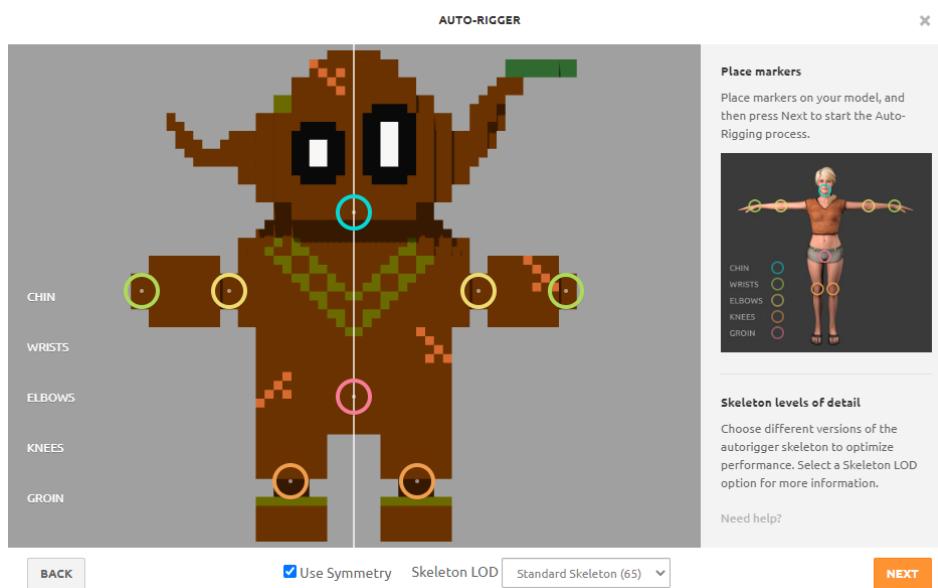


Figura 5.7: Definición del esqueleto del modelo

en este caso he escogido la animación *Mutant Walking*. Una vez finalizado, necesitaré 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 5.8: 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

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.

5.2.1. 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:

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

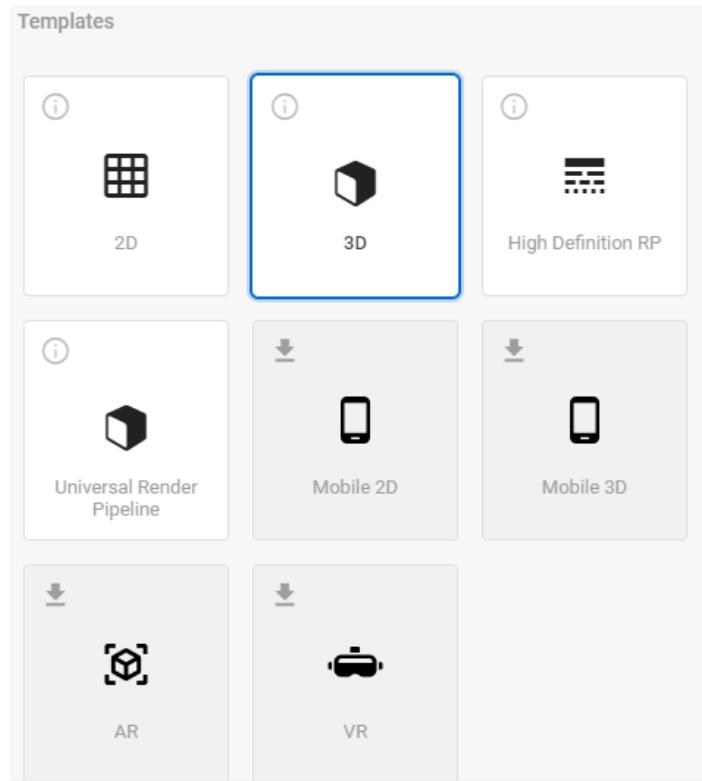


Figura 5.9: Tipos de proyecto disponibles en Unity

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.

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.

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 ésta se desarrollará en el siguiente apartado.

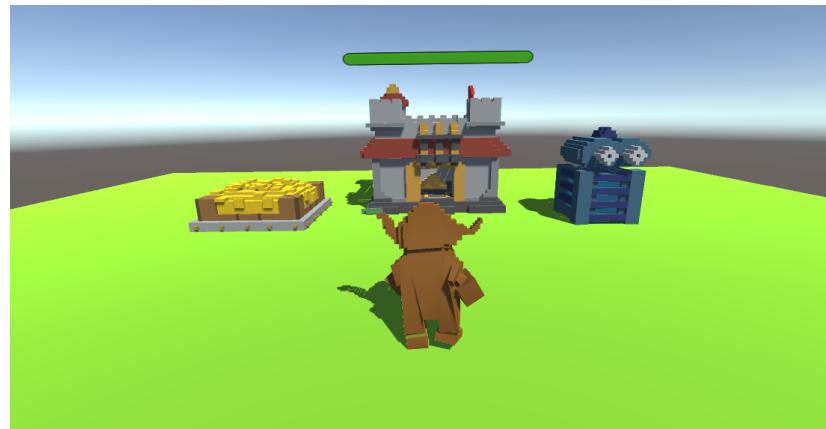


Figura 5.10: Escenario básico para el desarrollo del proyecto

5.2.2. 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 3.4). 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.

Los scripts que he creado para el desarrollo de este proyecto son los siguientes: ***FortalezaHealth.cs***, que se encarga de modificar el valor del slider que muestra la salud de la fortaleza, contiene la salud actual y máxima de esta y métodos para calcular y actualizar el valor del slider; ***AlmacenOro.cs***, realiza las mismas funciones que el script de la fortaleza pero indica la cantidad de oro en el almacén en lugar de su salud, contiene el dinero máximo y el actual y funciones para modificar el valor del slider; ***Torreta.cs***, que se encarga de controlar el comportamiento de la torreta haciendo que apunte siempre hacia el enemigo, contiene la posición del enemigo y una función para reducir su salud en función de la distancia a la que se encuentre; ***GoToGoal.cs***, es el script principal para el algoritmo de Aprendizaje por Refuerzo, contiene variables con la fortaleza y el almacén de oro, la salud máxima del enemigo y su salud actual, el dinero recaudado del almacén y una variable que indica si dispone del potenciador de ataque. Este script contiene además las funciones que controlarán al agente y decidirán que acciones toma como caminar o atacar a la fortaleza. Estas funciones se explicarán en profundidad en el siguiente apartado.

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

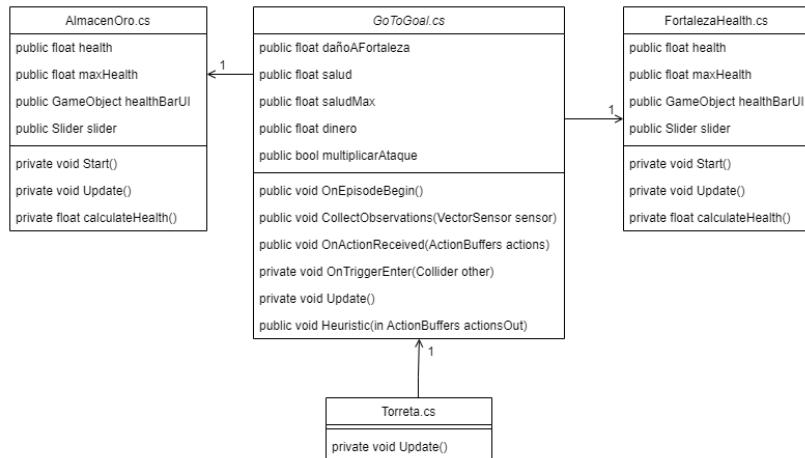


Figura 5.11: Diagrama de clases

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, desarrollo 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.

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:

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

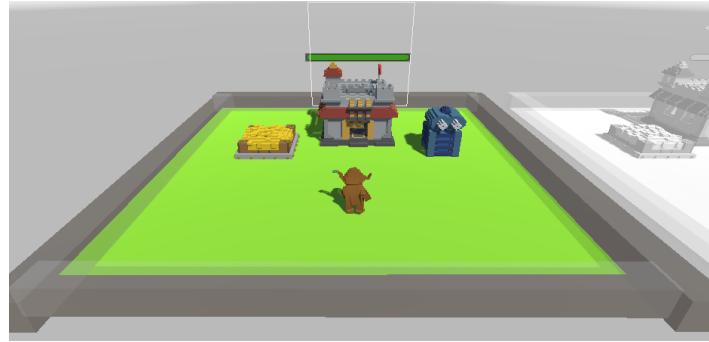


Figura 5.12: Escenario final para el desarrollo del proyecto

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 refugio. 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 5.13: 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 movi-

miento. 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:

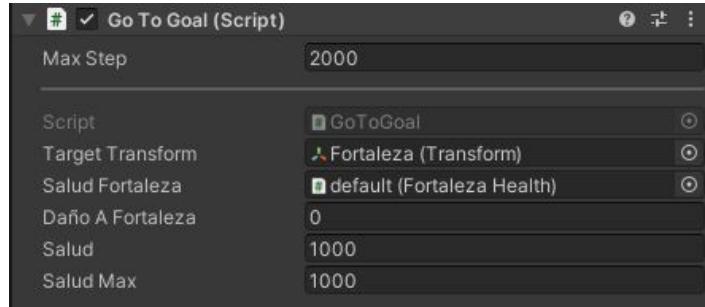


Figura 5.14: Variables en el editor de Unity

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: Torreta.cs

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.

Por último será necesario crear el script que indicará la cantidad de dinero del almacén de oro y modificar el algoritmo de Aprendizaje por Refuerzo para que se añada la opción de atacar el almacén de oro, obtener dinero y, al alcanzar el límite, obtener el potenciador de ataque. Antes de crear el script será necesario, al igual que para la fortaleza, crear un slider que indique la cantidad de oro que contiene el almacén y un collider para que el enemigo detecte las colisiones. Una vez asignado al almacén, el script es bastante sencillo de realizar ya que su estructura será muy similar al script que controla la salud de la fortaleza. De este modo contendrá el dinero actual y el máximo del almacén y una función para actualizar el estado del slider al igual que en `FortalezaHealth.cs`.

En el script `GoToGoal.cs` será necesario modificar las siguientes funciones y variables para que el agente pueda interaccionar con el almacén de oro.

```

public AlmacenOro saludAlmacenOro;
public float dinero;
public bool multiplicarAtaque;

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

    //Volvemos a inicializar la salud del enemigo, la
    fortaleza y el almacen

```

```

        salud = saludMax;
        saludFortaleza.health = saludFortaleza.maxHealth;
        saludAlmacenOro.health = saludAlmacenOro.maxHealth;

        //Pierde el dinero recaudado
        dinero = 0f;
        //Pierde las ventajas que tenia
        multiplicarAtaque = false;
    }

private void OnTriggerEnter(Collider other)
{
    if(other.TryGetComponent<Fortaleza>(out Fortaleza
        fortaleza))
    {
        [...]

        //Si ha llegado hasta el objetivo ataca
        if (multiplicarAtaque) //Si tiene la bonificacion
            de ataque
        {
            saludFortaleza.health -= 2f * Mathf.Abs(
                danoAFortaleza);
            AddReward(+2f*Mathf.Abs(danoAFortaleza));
        }
        else
        {
            saludFortaleza.health -= Mathf.Abs(
                danoAFortaleza);
            AddReward(+Mathf.Abs(danoAFortaleza));
        }

        [...]
    }
    if (other.TryGetComponent<Muro>(out Muro muro)) //Si
        se choca con un muro acaba el episodio
    {
        [...]
    }
    if (other.TryGetComponent<AlmacenOro>(out AlmacenOro
        almacen)) //Si se choca con un almacen de oro roba
        dinero
    {
        //Si ha llegado hasta el almacen ataca
        if(saludAlmacenOro.health>0){
            saludAlmacenOro.health -= Mathf.Abs(
                danoAFortaleza);
    }
}

```

```

        AddReward(+Mathf.Abs(danoAFortaleza));
        dinero += Mathf.Abs(danoAFortaleza);
    }
}

private void Update()
{
    [...]

    if (dinero >= saludAlmacenOro.maxHealth)
    {
        multiplicarAtaque = true;
    }
}

```

Extracto de código 5.8: Añadidos al agente para interaccionar con el almacén de oro

Veamos todos estos añadidos por partes. En primer lugar será necesario incluir una variable que contenga al almacén, una variable que contenga el dinero recogido y una variable que indique si el enemigo ha adquirido el multiplicador de ataque. Además, habrá que inicializar las variables correctamente cada vez que el episodio comience en la función *OnEpisodeBegin()*. En la función *OnTriggerEnter()* será necesario modificar la sección para dañar a la fortaleza, de modo que si el enemigo dispone del multiplicador de ataque, el daño que esta reciba sea doble. Incluyo además otra sección en esta función para que cuando el agente colisione con el almacén este reciba daño y el enemigo gane dinero. En la función update comprobaré si el enemigo tiene suficiente dinero y, cuando sea posible, recibirá el multiplicador de ataque.

Una vez completado esto, el algoritmo de Aprendizaje por Refuerzo está listo para ser entrenado. Sin embargo, el agente corre el riesgo de aprender únicamente que debe moverse a una posición concreta del mapa en lugar de aprender a atacar a la fortaleza y el almacén y evitar la torreta. Por tanto, en el inicio de cada episodio generaré la posición de cada una de estas estructuras aleatoriamente dentro de un rango determinado. Para ello, simplemente añado también el almacén de oro y la torreta al agente en sus observaciones y genero el valor aleatorio de la posición en la función *OnEpisodeBegin()* mediante la función *Random.Range(x,y)*, que genera un valor aleatorio comprendido entre x e y. En el siguiente apartado realizaré las pruebas en las que entrenaré el algoritmo y compararé los resultados obtenidos para comprobar la eficiencia de este método.

5.3. Pruebas

En este capítulo realizaré diversas pruebas, entrenando el algoritmo de Aprendizaje por Refuerzo para distintos valores de recompensas. Posteriormente, compararé los distintos resultados obtenidos en los entrenamientos para estos valores, así como

los resultados obtenidos en los entrenamientos que realicé conforme implementaba el algoritmo añadiendo más funcionalidades.

5.3.1. Evolución del entrenamiento durante el proceso de desarrollo

Veamos las distintas gráficas obtenidas durante el proceso de entrenamiento a lo largo del proceso de desarrollo del algoritmo de Aprendizaje por Refuerzo. Las gráficas muestran los resultados de los entrenamientos realizados para: el enemigo únicamente aprende a atacar la fortaleza (**pruebaAtacarFortaleza**), el enemigo intenta atacar la fortaleza pero también recibe daño de la torreta (**pruebaAtacarConTorreta**), el enemigo ataca tanto a la fortaleza como al almacén de oro con la torreta disparándole (**pruebaAlmacenOro**) y, por último, el caso anterior pero las posiciones de las distintas estructuras se generan aleatoriamente (**pruebaLocalizacionAleatoria**). Los colores asociados a cada una de las pruebas se muestran a continuación:

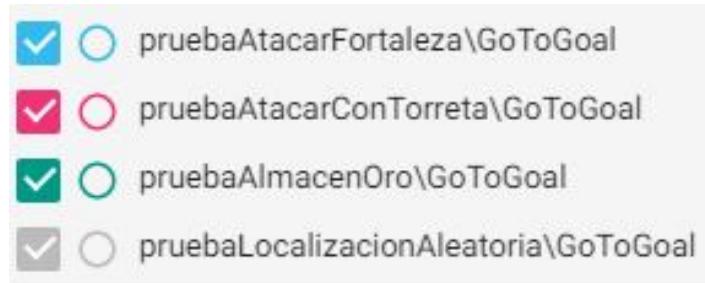


Figura 5.15: Leyenda para los entrenamientos realizados durante el desarrollo

Las gráficas con los resultados obtenidos se clasificarán en tres categorías: entorno, pérdidas y política.

Dentro de la sección de entorno podremos observar una gráfica asociada a la recompensa acumulada (**Cumulative Reward**), que mostrará la recompensa acumulada media obtenida por todos los agentes (en mi caso se trata de un sólo agente). Si la sesión de entrenamiento tiene éxito esta gráfica debería aumentar. Dentro de este apartado se encuentra también la gráfica con la longitud de los episodios (**Episode Length**), que mostrará la duración media de cada episodio en el entorno.

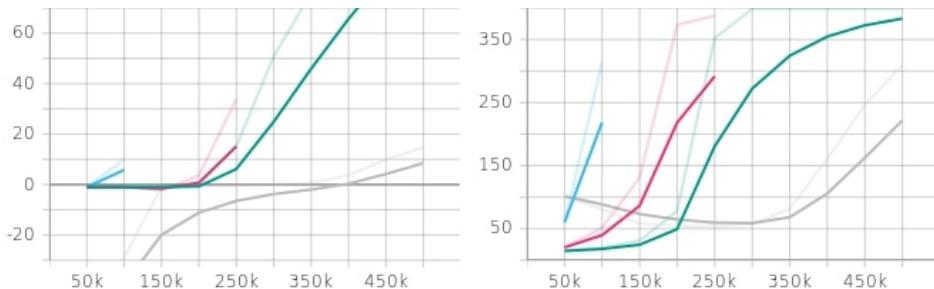


Figura 5.16: Cumulative Reward y Episode Length

Como podemos observar en la figura 5.16, el comportamiento de cada gráfica actúa según lo esperado. A pesar de que los dos primeros test no se ejecutaron tantas veces como los otros podemos observar el comportamiento de las gráficas en estos casos también. En la gráfica del *Cumulative Reward*, todas las gráficas son crecientes la mayor parte del tiempo, exceptuando algunos puntos en los que la recompensa se reduce en casos como el de la torreta (rosa), debido probablemente a que el personaje se encontraba más próximo a la torreta en algunos casos o salía del escenario. Sin embargo, el entrenamiento avanza correctamente en todos los casos, aumentando siempre la media de recompensas acumuladas. Cuando los elementos empezaron a generarse aleatoriamente podemos ver que el tiempo en el que la recompensa media se mantenía negativa era mayor, lo cual era de esperar debido a la mayor dificultad que supondrá para la IA aprender a esquivar todos los elementos perjudiciales para esta. Del mismo modo, podemos ver que la gráfica de *Episode Length* también es creciente en el tiempo como cabía esperar a excepción de la gráfica gris correspondiente al entrenamiento cuando los elementos se generaban aleatoriamente. Esto puede deberse a que el enemigo empieza intentando evitar a la torreta, que cada vez se genera en un lugar aleatorio, y accidentalmente se sale del escenario o acaba acercándose a la nueva posición de la torreta en el siguiente caso. A pesar de esto podemos ver que la duración de los episodios empieza a aumentar, coincidente con el punto en el que las recompensas empiezan a ser positivas.

En la sección de pérdidas podemos encontrar otras dos gráficas. La primera mostrará la pérdida de la política (**Policy Loss**), que contiene el valor medio de la función de pérdida de la política. Esta gráfica está relacionada con cuánto cambia la política (el proceso para tomar decisiones), por lo que, si la sesión de entrenamiento es exitosa, la gráfica debería decrecer. La segunda gráfica dentro de este apartado es la gráfica sobre la pérdida de valor (**Value Loss**), que indica la pérdida media de la actualización de la función de valor. Se relaciona con lo bien que el modelo es capaz de predecir el valor de cada estado. Así, esta función debería aumentar conforme el agente aprende y, una vez que la recompensa se estabiliza, debería disminuir.

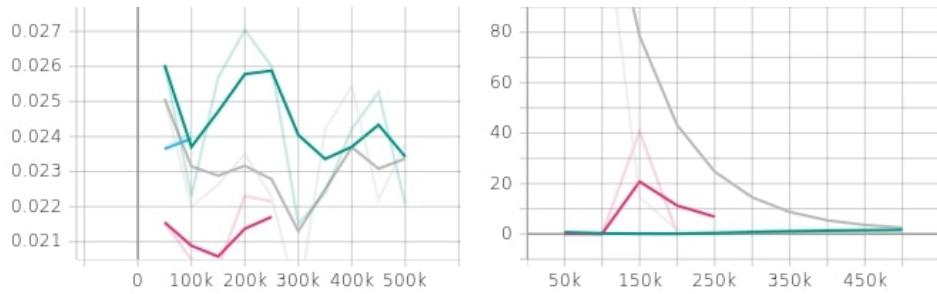


Figura 5.17: Policy Loss y Value Loss

En este caso no podemos obtener muchas conclusiones al observar las gráficas sobre los dos primeros entrenamientos por lo que centraré el análisis en los dos últimos, que corresponden a las gráficas verde y gris. Al observar la gráfica *Policy Loss* podemos ver que ambas gráficas se mantienen irregulares, decreciendo en algunas ocasiones y creciendo en otras. Este comportamiento no es del todo extraño, sobre todo en el caso de la gráfica gris, que corresponde al caso en el que los elementos se generan en posiciones aleatorias, ya que una política que puede funcionar en varias ocasiones seguidas puede

variar de repente si los elementos se generan en posiciones que perjudican al enemigo, obligando a calcular una nueva política. A pesar de esto, ambas gráficas no crecen en demasiadas ocasiones por lo que el aprendizaje podría considerarse exitoso en su mayor parte. La gráfica *Value Loss* muestra que la pérdida media cuando añadí el almacén de oro se mantuvo constante la mayor parte del tiempo, mientras que en el caso de la localización aleatoria de los elementos es constantemente decreciente, probablemente debido a la estabilización de la recompensa en este caso.

La sección de política es la que incluye un mayor número de gráficas. En primer lugar tenemos la función **Beta**, que indica el nivel de exploración a lo largo del entrenamiento por lo que, si el entrenamiento tiene éxito, se reducirá con el tiempo. La gráfica de entropía (**Entropy**), indica cómo de aleatorias son las decisiones que toma el modelo por lo que debería reducirse lentamente con el tiempo. La recompensa extrínseca (**Extrinsic Reward**) corresponde a la recompensa media acumulada recibida por episodio, mientras que el valor extrínseco estimado (**Extrinsic Value Estimate**), es el valor medio estimado para todos los estados visitados por el agente (debería ser una función creciente). Por último, la tasa de aprendizaje (**Learning Rate**), indica cómo de largo es el paso que tiene que dar el algoritmo para encontrar la política óptima, por ello, debería disminuir con el tiempo.

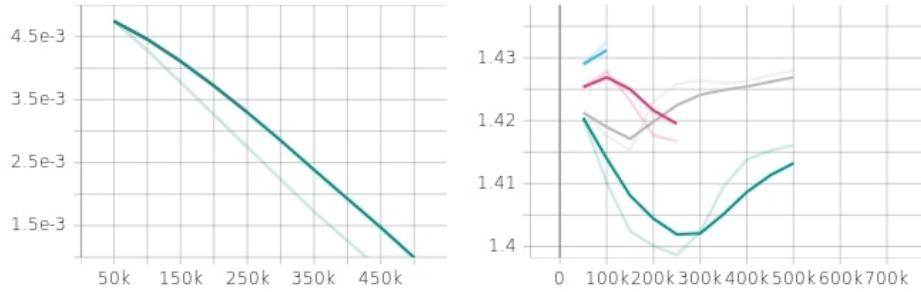


Figura 5.18: Beta y Entropy

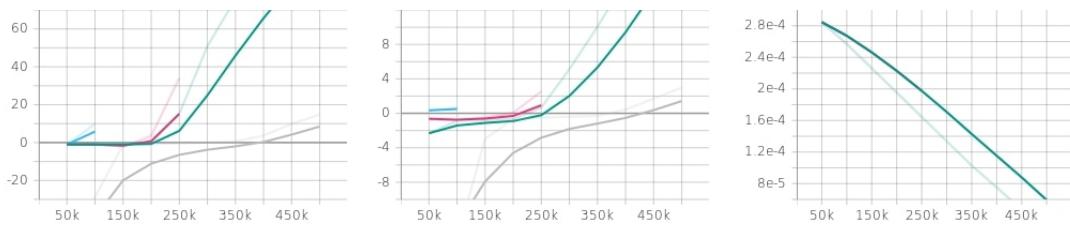


Figura 5.19: Extrinsic Reward, Extrinsic Value Estimate y Learning Rate

Como podemos observar en la figura 5.18, todas las gráficas se superponen en la gráfica Beta, por lo que el comportamiento en todos los entrenamientos ha sido el esperado y la exploración se reduce según avanza el entrenamiento. En la gráfica Entropy, podemos observar que todas las gráficas comienzan descendiendo, indicando que la aleatoriedad de las decisiones se reduce con el paso del tiempo. Sin embargo, en el caso de las gráficas verde y gris (almacen de oro y posiciones aleatorias respectivamente), la entropía crece hacia el final del entrenamiento, debido probablemente a que el enemigo recibe recompensas tanto de la fortaleza como del almacén de oro y que, en el caso de la gráfica gris, las posiciones de las estructuras son aleatorias en todo momento.

En la figura 5.19, podemos observar un comportamiento esperable tanto en las gráficas Extrinsic Reward y Extrinsic Value Estimate, que tienen una forma parecida y coinciden con los valores de la gráfica del Cumulative Reward que hemos observado en la figura 5.16. Por último, la gráfica del Learning Rate también tiene un comportamiento esperado, decreciendo con el paso del tiempo, lo que implica que cada vez el paso para encontrar la política óptima es más corto.

Puedo concluir por tanto, que los entrenamientos realizados durante el proceso de desarrollo han sido exitosos, por lo que procedo a entrenar el algoritmo final mediante distintas recompensas para comprobar qué método es más eficaz y me aporta mejores resultados.

5.3.2. Valores de las recompensas

En la siguiente tabla muestro los valores de las recompensas asociadas a cada entrenamiento realizado. La columna de fortaleza indicará la recompensa que el agente obtendrá por atacar la fortaleza, la columna de almacén indica la recompensa que el enemigo obtiene al golpear el almacén y la columna de torreta indica la penalización que este recibirá al ser atacado por la torreta. La penalización por salir del mapa siempre será de menos uno.

Lista de entrenamientos			
Entrenamiento	Fortaleza	Almacen	Torreta
1	dañoFortaleza	dañoAlmacen	dañoTorreta
2	dañoFortaleza	dañoAlmacen	-0.1
3	dañoFortaleza	0.5	dañoTorreta
4	1.0	dañoAlmacen	dañoTorreta
5	1.0	0.5	-0.1

Cuadro 5.2: Listado de entrenamientos y recompensas asociadas

Como se puede observar en la tabla, realizaré un entrenamiento en el que las recompensas que reciba el enemigo varíen en función del daño que este haga a la fortaleza y el almacén y el daño que reciba de la torreta. Después, realizaré entrenamientos en los que asigne un valor concreto a cada una de estas variables. La penalización de la torreta será de -0.1 debido a que el enemigo puede recibir daño constantemente de la torreta por lo que no necesita ser un valor muy alto. Las recompensas por atacar el almacén y la fortaleza serán de 0.5 y 1.0 respectivamente, ya que el objetivo principal será reducir la salud de la fortaleza. Por último, realizaré un entrenamiento en el que todas las recompensas sean valores fijos y compararé los resultados obtenidos en cada uno.

5.3.3. Análisis de los resultados obtenidos

En primer lugar, mostraré la leyenda con los colores que representan a cada entrenamiento realizado:

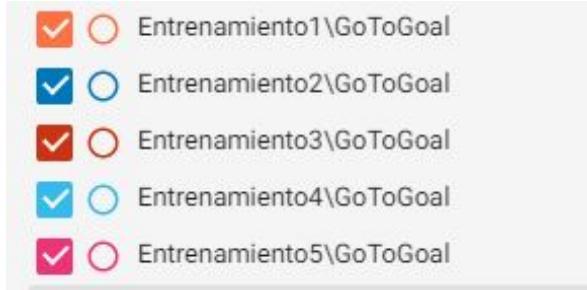


Figura 5.20: Leyenda para los entrenamientos finales

Procedo ahora a observar y comparar los resultados obtenidos en las distintas gráficas. En primer lugar, comparemos los resultados obtenidos en la sección de entorno. Las gráficas obtenidas pueden observarse en la siguiente figura:

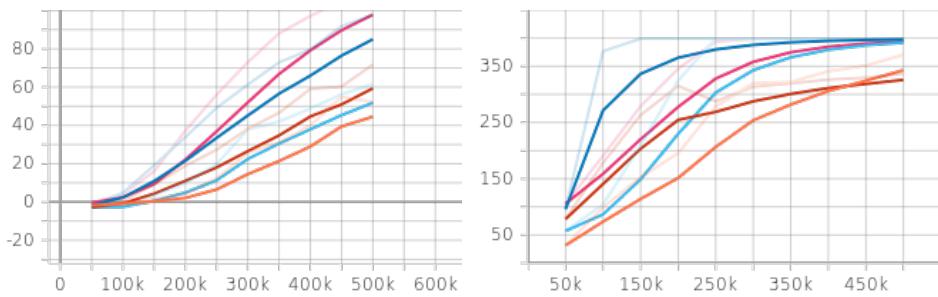


Figura 5.21: Cumulative Reward y Episode Length (Prueba final)

Podemos observar en ambos casos que, a simple vista, los entrenamientos parecen haberse ejecutado con éxito debido a que en ambos casos encontramos funciones constantemente crecientes. En el caso de la gráfica *Cumulative Reward* parece que los entrenamientos 2 y 5 son los que mayores recompensas a largo plazo han obtenido, habiendo una diferencia de casi 20 puntos con el tercer entrenamiento más exitoso en este aspecto. Además, en la gráfica *Episode Length*, podemos comprobar que el segundo entrenamiento ha sido el que ha tenido un crecimiento más rápido. Esto nos muestra que el **entrenamiento 2** es el que más rápido ha aprendido a obtener una recompensa alta y lograr que los episodios duren más en el menor tiempo de entrenamiento, por lo que, por ahora, parece ser el entrenamiento más prometedor. Sin embargo, los entrenamientos 4 y 5 también han conseguido alcanzar la duración máxima de los episodios cerca del final del entrenamiento, por lo que será importante observarlos con atención. Por ahora, los entrenamientos 1 y 3 parecen ser los menos prometedores, obteniendo resultados más pobres que el resto en ambos apartados.

Mostraré a continuación los resultados obtenidos en la sección de pérdidas para comparar el rendimiento de los cinco entrenamientos. En esta sección podremos observar la pérdida de la política, así como la pérdida de valor. Los resultados pueden observarse en la siguiente figura.

Centrémonos primero, en la gráfica *Policy Loss*. Como comenté anteriormente, esta gráfica debería decrecer con el tiempo si la sesión de entrenamiento es exitosa ya que está directamente relacionada con cuánto cambia la política. Aunque todos los entrenamientos parecen tener un comportamiento similar, creciendo bastante la

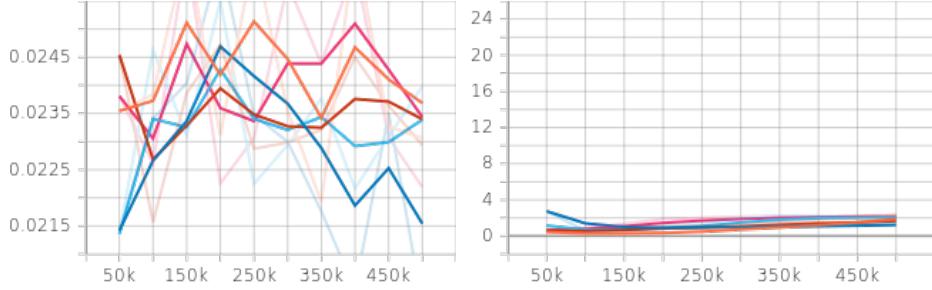


Figura 5.22: Policy Loss y Value Loss (Prueba final)

gráfica al principio de este y luego decreciendo y creciendo de manera algo más suave, podemos ver de nuevo que el segundo entrenamiento parece comportarse mejor que los demás, disminuyendo a partir del valor 200k aproximadamente y creciendo levemente únicamente al final del entrenamiento. De la gráfica *Value Loss* no obtenemos resultados muy importantes, ya que todos los entrenamientos tienen un comportamiento similar bastante estable causado por el hecho de que consiguen estabilizar rápidamente la recompensa que obtienen.

Por último, comparemos los resultados obtenidos en la sección de política. Los resultados obtenidos se muestran a continuación.

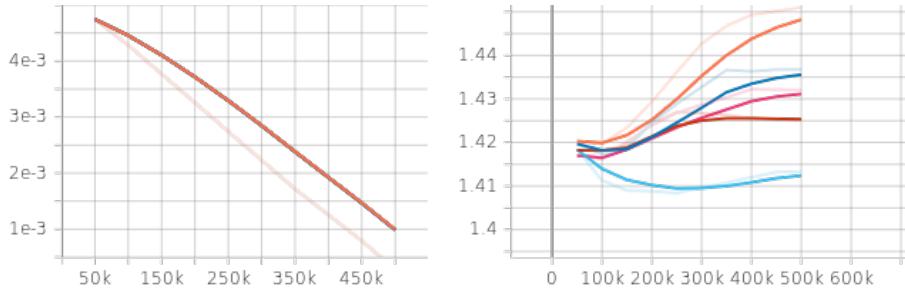


Figura 5.23: Beta y Entropy (Prueba final)

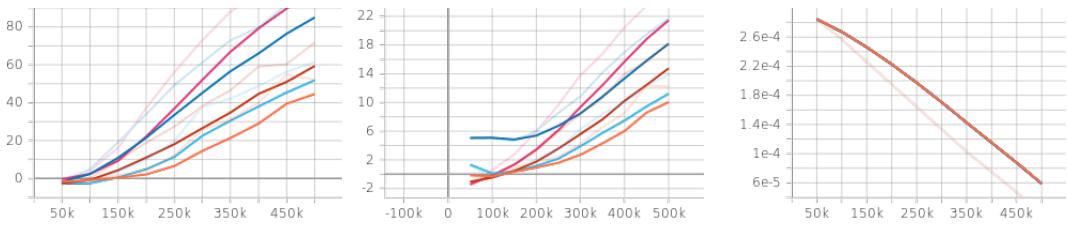


Figura 5.24: Extrinsic Reward, Extrinsic Value Estimate y Learning Rate (Prueba final)

Como se puede observar, las gráficas *Beta* y *Learning Rate* no nos aportan mucha información sobre cuál de los entrenamientos es mejor, ya que en todos ellos se obtiene un resultado prácticamente idéntico y esperable (al igual que analicé en el apartado de los resultados obtenidos en el proceso de desarrollo). Observemos entonces la gráfica *Entropy*. Esta gráfica nos muestra cómo de aleatorias son las decisiones que toma el algoritmo, por lo que debería reducirse y estabilizarse con el tiempo. Sorprendentemente,

aunque todas varían mínimamente dentro de unos rangos muy pequeños, el entrenamiento que se mantiene más estable es el cuarto, el cual consiguió alcanzar también la longitud máxima de episodio, como comenté anteriormente. Por último, las gráficas *Extrinsic Reward* y *Extrinsic Value Estimate* muestran resultados bastante similares a los aportados por *Cumulative Reward*, destacando de nuevo los entrenamientos 2 y 5, cambiando el orden de estos sin embargo.

A partir de los datos recolectados, podemos concluir que los entrenamientos que más éxito han tenido han sido los entrenamientos 2 y 5, destacando sobre todo el segundo entrenamiento en la mayoría de aspectos. Para finalizar el proyecto, aplicaré la IA entrenada por cada uno de estos métodos en el enemigo y observaré si esta IA es verdaderamente capaz de dañar a la fortaleza de manera eficaz.

5.3.4. Aplicación de las IAs obtenidas

En este apartado, ejecutaré 10 veces el programa con cada una de las IAs entrenadas y obtenidas en los entrenamientos del apartado anterior. Para compararlas, ya que el objetivo principal del juego era reducir lo máximo posible la salud de la fortaleza, observaré la salud final de la fortaleza antes de que el enemigo muera o el episodio acabe.

Para obtener la IA entrenada, basta con acceder al directorio generado con los resultados del entrenamiento. Allí se encuentra el archivo de tipo *.onnx* que contiene a la IA entrenada. Para asignársela al enemigo, basta con arrastrar este archivo hasta la sección Model y luego seleccionar como tipo de comportamiento (Behaviour Type) el tipo *Inference Only*.

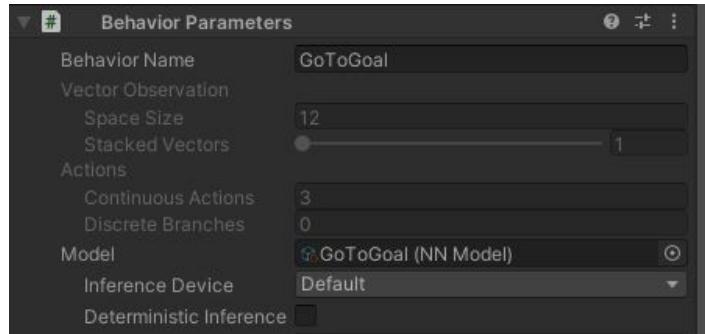


Figura 5.25: Asignar la IA al enemigo creado

A continuación, mostraré una tabla con los resultados obtenidos para cada uno. En las columnas se encuentra cada entrenamiento con la vida final de la fortaleza al final de cada ejecución, así como la media de los valores obtenidos. La vida máxima de la fortaleza en cada ejecución es siempre de 100.

Durante las ejecuciones he podido observar que el **primer entrenamiento** ha dado lugar a una IA que evita la torreta y, a la vez, intenta atacar tanto el almacén de oro como la fortaleza, por eso obtiene resultados bastante uniformes. El **segundo entrenamiento** (que era el más prometedor al estudiar los resultados obtenidos en el entrenamiento), por otro lado, ha dado lugar a una IA más arriesgada que no da

Resultados de los entrenamientos					
Ejecución/Entrenam	1	2	3	4	5
1	81.06	67.11	99.19	91.03	87.45
2	68.89	79.04	73.83	90.03	89.71
3	78.37	65.81	100	97.36	84.94
4	69.83	50.22	58.12	100	75.92
5	68.25	56.14	99.40	94.85	74.38
6	94.57	91.15	100	96.19	94.60
7	95.79	88.63	54.03	100	94.18
8	92.24	99	100	92.22	94.91
9	83.04	85.09	63.98	97.94	89.08
10	96.05	80.89	67.39	99.80	85.12
Media	82.81	76.30	81.59	95.94	87.03

Cuadro 5.3: Salud de la fortaleza tras ejecutar la IA obtenida en cada entrenamiento

importancia a la torreta y se centra más en atacar la fortaleza. Por esto mismo, en algunas ejecuciones consigue hacer una gran cantidad de daño a la fortaleza pero, en otras, apenas consigue llegar a ella por no evitar la torreta. El **tercer entrenamiento**, por su parte, ha dado lugar a una inteligencia más patosa que intenta centrarse en la fortaleza sin mucho éxito en la mayoría de ocasiones. Esto tiene sentido observando los resultados obtenidos durante el entrenamiento. El **cuarto entrenamiento** ha dado lugar a una IA similar a la del primer entrenamiento aunque, a diferencia de este, parece no tener tan en cuenta a la torreta por lo que obtiene peores resultados generalmente. Por último, el **quinto entrenamiento** ha dado lugar a una IA similar a las 3 anteriores ya que se centra más en atacar la fortaleza pero obtiene mejores resultados evitando la torreta.

Independientemente de los resultados obtenidos en las medias, he podido observar que muchas de las IAs generadas tienden a tomar el camino derecho para atacar la fortaleza independientemente de dónde se genere la torreta. Esto puede deberse a que durante el entrenamiento la torreta se genera principalmente en zonas centrales o izquierdas y que el refuerzo negativo por recibir daño de esta no sea lo suficientemente alto. Además, ninguna parece sacar demasiado provecho del bonus de ataque que se obtiene al conseguir el oro del almacén, quizás debido al poco tiempo del que dispone el enemigo o que el refuerzo obtenido al atacar el almacén no es suficientemente grande.

Si observamos las medias obtenidas en la tabla veremos, como había deducido en el apartado anterior, que el algoritmo de Aprendizaje por Refuerzo que obtiene mejores resultados es el obtenido en el segundo entrenamiento. Esto se debe, como he comentado antes, debido a que genera una IA más temeraria que centra toda su atención en atacar la fortaleza y, aunque a veces consigue resultados muy malos en comparación con la media de las otras IAs obtenidas, generalmente consigue obtener resultados muy buenos atacando la fortaleza.

Realizo, por último, un modelo tipo ANOVA para comparar si las 5 Inteligencias Artificiales obtenidas presentan diferencias en la media de la salud final de la fortaleza. Para ello, utilizaré *Rstudio*, creando en primer lugar un dataframe, formado por un vector que contiene la salud final de la fortaleza, y otro que contiene la IA ejecutada.

Realizo, a continuación, el ANOVA mediante la función `aov(salud IA, data=dataframe)`. La tabla ANOVA asociada a estos datos es la siguiente:

	G1	Suma Cuadrados	Media Cuadrados	F	P-valor
IA	4	2154	538.6	3.202	0.0214
Residual	45	7571	168.2		

Cuadro 5.4: Tabla ANOVA

El test asociado a esta tabla es el siguiente:

$$\begin{cases} H_0 : \mu_1 = \dots = \mu_5 \\ H_1 : \mu_i \neq \mu_j, \text{ para algún } i, j \in \{1, 2, 3, 4, 5\} \end{cases}$$

El p-valor obtenido es menor que el nivel de significación $\alpha = 0,05$, por lo que se rechaza la hipótesis nula de homogeneidad de medias entre las Inteligencias Artificiales. Esto significa que debe de haber diferencias entre algunas de las medias.

Para ver estas diferencias, usaré la función `TukeyHSD` para hacer las comparaciones múltiples entre las medias. Este método se basa en calcular un intervalo de confianza para la diferencia entre las medias y un p-valor asociado. Si este p-valor resulta ser menor que el nivel de significación α se rechaza la hipótesis nula, que es de igualdad entre las medias.

	diff	lwr	upr	p valor
2-1	-6.501	-22.98	9.98	0.79
3-1	-1.215	-17.69	15.26	0.99
4-1	13.13	-3.34	29.61	0.17
5-1	4.22	-12.26	20.70	0.94
3-2	5.28	-11.19	21.76	0.89
4-2	19.63	3.15	36.11	0.01
5-2	10.72	-5.76	27.20	0.35
4-3	14.34	-2.13	30.83	0.11
5-3	5.43	-11.04	21.91	0.88
5-4	-8.91	-25.39	7.56	0.54

Cuadro 5.5: Tabla ANOVA

En todos los test asociados a la tabla no se rechaza la igualdad de medias, a excepción de los grupos 4 y 2 (con p-valor=0.012). Por tanto, no existe una diferencia significativa en media entre la salud final de la fortaleza obtenida en el resto de casos. Este resultado tiene sentido ya que el segundo entrenamiento es el que ha dado lugar a la IA más prometedora, mientras que el cuarto entrenamiento es el que ha provocado peores resultados.

Para que los resultados de este método sean válidos es necesario comprobar las hipótesis de partida. Las más importantes son la normalidad de los residuos y la homogeneidad de varianzas entre grupos. Para evaluar la normalidad de los residuos represento mediante un gráfico `qqnorm` (cuantil cuantil normal) los cuantiles de la distribución de la salud frente a los teóricos de la normal. Como podemos observar en el

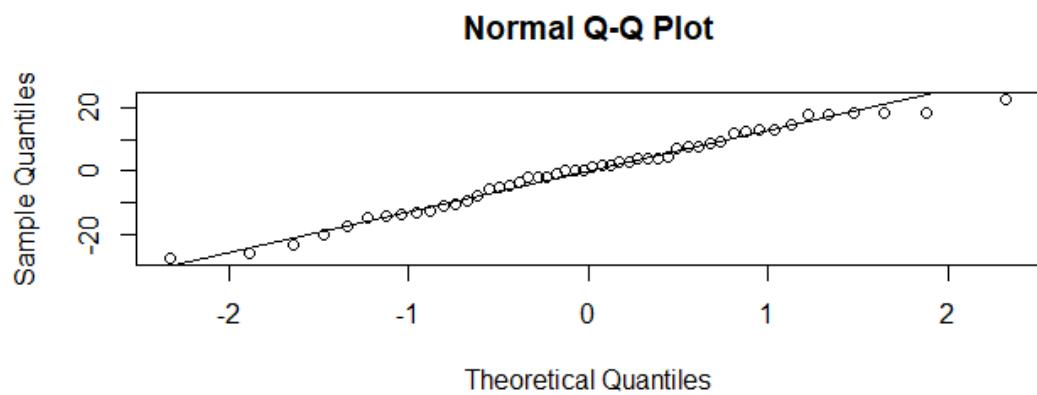


Figura 5.26: Gráfico Q-Q Plot

siguiente gráfico, los puntos se encuentran prácticamente alineados, por lo que no hay ninguna razón para sospechar que no sean normales.

Para la homogeneidad de varianzas entre grupos empleo el test de Barlett, cuya hipótesis nula es la homogeneidad de varianzas. El p-valor obtenido al realizar el test es 0.000014, por lo que a una confianza del 95 % se rechaza la homogeneidad de varianzas.

6. Conclusiones y vías futuras

Como he podido comprobar, al aplicar métodos de Aprendizaje por Refuerzo para entrenar a la Inteligencia Artificial que empleará un agente en un videojuego, podemos obtener IAs variadas simplemente modificando los parámetros de las recompensas que reciben los avatares al realizar diferentes acciones.

De este modo, como en el caso del segundo entrenamiento que he realizado, es posible obtener enemigos que sean capaces de hacer bastante daño a la fortaleza a pesar de que puedan tomar decisiones más arriesgadas en determinadas ocasiones. Así mismo, podríamos obtener en un futuro (por ejemplo) avatares que tuviesen mayor salud, menos velocidad de movimiento y cuya prioridad fuese atacar el almacén de oro, o también sería posible crear enemigos más rápidos y con menos salud cuya prioridad fuese evitar las torretas a toda costa. Todo esto podría lograrse utilizando un algoritmo similar al que he desarrollado, modificando principalmente los valores de las recompensas, para obtener IAs más variadas.

Como comenté en la introducción del proyecto, algunas empresas ya están trabajando en realizar Inteligencias Artificiales para conseguir crear experiencias de juego más variadas. Este método no sólo permitirá obtener enemigos más avanzados que supondrán un mayor reto al jugar ya que, como he comentado anteriormente, también permitirá obtener distintos tipos de enemigos modificando los valores de las recompensas que cada uno reciba por realizar una determinada acción. Esto supondría una mejora en la velocidad de producción ya que reduciría la cantidad de código a programar, siendo posible con un mismo algoritmo entrenar a distintos tipos de avatares.

Sin embargo, un aspecto que sería necesario estudiar en un futuro, serían los resultados que se obtienen aplicando para estos métodos a avatares que interactúen con el jugador, comprobando si los resultados muestran una gran diferencia frente a los obtenidos en este proyecto. Otro campo interesante de estudio sería aplicar este método para crear una IA que ayudase al jugador, en lugar de enfrentarse a él, ya que en este caso la inteligencia debería elegir entre atacar la fortaleza (suponiendo un escenario similar al planteado en este proyecto) y ayudar al jugador cuando esté en una situación de riesgo.

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] Domestika. Artículo sobre qué es el voxel art, 2020. URL <https://www.domestika.org/es/blog/5529-que-es-el-voxel-art>.
- [4] Yuanheng Zhu Kun Shao, Zhentao Tang. A survey of deep reinforcement learning in video games, 2019. URL <https://arxiv.org/abs/1912.10944>.
- [5] Unity Technologies. Página principal de unity, 2020. URL <https://unity.com/es>.
- [6] Unity Technologies. Repositorio en github de ml-agents, 2022. URL <https://github.com/Unity-Technologies/ml-agents>.
- [7] Unity Technologies. Ml-agents en la página oficial de unity, 2022. URL <https://unity.com/es/products/machine-learning-agents>.
- [8] Santiago Bidegain y Pablo García Sánchez. Performance study of minimax and reinforcement learning agents playing the turn-based game iwoki, 2021. URL <https://www.tandfonline.com/doi/full/10.1080/08839514.2021.1934265>.
- [9] Georgios N. Yannakakis and Julian Togelius. In *Artificial Intelligence and Games*, pages 71–77, 2018.