Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Agentes inteligentes compitiendo contra un jugador real en mapas generados aleatoriamente en Unity

Intelligent agents competing against a real player on randomly generated maps in Unity

Pablo Gil de San Nicolás

Universidad de La Laguna
Grado en Ingeniería Informática
La Laguna, Julio de 2024

D. **Rafael Arnay del Arco,** profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

“Agentes inteligentes compitiendo contra un jugador real en mapas generados aleatoriamente en Unity”

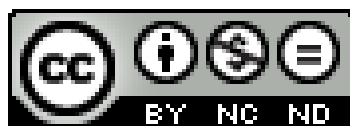
ha sido realizada bajo su dirección por D. **Pablo Gil de San Nicolás**.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de julio de 2024

Agradecimientos

En primer lugar, querría agradecer a mi tutor de TFG, Rafael Arnay del Arco, por guiarme a lo largo del desarrollo del proyecto, y ayudarme a continuar en los momentos en los que se me hacía complicado avanzar.

Además, especial agradecimiento a mi familia y amigos, que se mantuvieron a mi lado y me brindaron todo su apoyo para que fuera capaz de llevar a cabo, no solo este proyecto, sino toda mi carrera.



© Esta obra está bajo una licencia de Creative Commons
Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional.

Resumen

En los últimos años, han surgido técnicas de Inteligencia Artificial (IA) basadas en redes neuronales, utilizadas en muchas aplicaciones para la generación automática de contenido: textos e imágenes, modelos 3D, música...

El objetivo de este trabajo ha sido la investigación de una de estas técnicas, el Aprendizaje por Refuerzo (Reinforcement Learning, RL), en el ámbito de los videojuegos para el desarrollo de comportamientos de Personajes no Jugador (PNJ) en entornos generados aleatoriamente para que compitan contra el jugador en una serie de actividades.

Este trabajo se ha implementado con el motor gráfico Unity [1], su entorno de desarrollo y el kit de herramientas ML-Agents [2].

Por otro lado, para la generación de los mapas se ha escogido el algoritmo de Wave Function Collapse [3], que tanto su versión original como su implementación en Unity [4] son de código abierto.

Palabras clave: Wave Function Collapse, ML-Agents, Machine Learning, Aprendizaje por Refuerzo, Unity

Abstract

In the last few years, techniques of Artificial Intelligence (AI) based on neural networks have emerged, used in many applications for the automatic generation of content: texts and images, 3D models, music...

The aim of this work is to research one of these techniques, Reinforcement Learning (RL), in the field of video games for the development of Non-Player Characters (NPCs) behaviours in randomly generated environments to compete against the player in a series of activities.

This work has been implemented with the Unity graphics engine [1], its development environment, and the ML-Agents toolkit [2].

On the other hand, for the generation of maps, the Wave Function Collapse algorithm [3] has been chosen, both its original version and its implementation in Unity [4] are open source.

Keywords: Wave Function Collapse, ML-Agents, Machine Learning, Reinforcement Learning, Unity

Índice general

Capítulo 1 Introducción **1**
1.1 Antecedentes 2
Capítulo 2 Desarrollo del proyecto **3**
2.1 Wave Function Collapse 3
2.1.1 Desarrollo de los tipos de casilla 5
2.1.2 Guardado y carga de mapas 8
2.2 ML-Agents 9
2.2.1 El aprendizaje por refuerzo 9
2.2.2 Componentes clave de ML-Agents 10
2.2.3 Componentes del entorno de entrenamiento 12
2.2.4 Proximal Policy Optimization 13
Alcanzar el objetivo 14
Presa contra depredador 15
Galería de tiro 20
Capítulo 3 Resultados obtenidos **23**
3.1 Juego de caza 23
3.1.1 Entrenamiento en escenario simple 24
3.1.2 Entrenamiento en escenario de Wave Function Collapse 26
3.1.3 Partidas simuladas 28
3.2 Tiempos de entrenamiento 29
Capítulo 4 Conclusiones y líneas futuras **31**
Capítulo 5 Summary and Conclusions **32**
Capítulo 6 Presupuesto **33**
6.1 Apartados 33
Capítulo 7 Bibliografía **34**
Capítulo 8 Anexo **36**

Índice de figuras

Figura 2.1: *Ejemplo de mapa 3D generado con Wave Function Collapse [3]* 3
Figura 2.2: *Patrones de vecindad automáticos [3]* 4
Figura 2.3: *Ejemplo generado con assets del proyecto base* 4
Figura 2.4: *Modelo de celda “Floor”* 5
Figura 2.5: *Modelo de celda “Wall”* 5
Figura 2.6: *Modelo generado con WFC, mayor densidad de muros* 6
Figura 2.7: *Modelo generado con WFC, menor densidad de muros* 6
Figura 2.8: *Modelo de celda “Grass”* 7
Figura 2.9: *Matriz de capas de colisión* 7
Figura 2.10: *Modelo generado con WFC con tres tipos de casillas* 8
Figura 2.11: *Botones en el inspector para el guardado y carga de mapas* 9
Figura 2.12: *Proceso de decisión Markov [14]* 10
Figura 2.13: *Componentes clave de ML-Agents [14]* 11
Figura 2.14: *Componentes del entorno de entrenamiento [14]* 12
Figura 2.15: *Escenario de alcanzar al objetivo* 14
Figura 2.16: *Parámetros de comportamiento del primer agente* 14
Figura 2.17: *Recompensas acumuladas del agente que se mueve hacia un objetivo* 15
Figura 2.18: *Sensor de percepción de rayos del agente depredador* 16
Figura 2.19: *Primera iteración del escenario presa contra depredador, mostrándose solamente el agente presa y su objetivo* 16
Figura 2.20: *Sensor de percepción de matriz del agente presa* 18
Figura 2.21: *Representación visual del sensor de percepción de matriz* 18
Figura 2.22: *Iteración final del escenario presa contra depredador* 19
Figura 2.23: *Recompensas acumuladas durante el entrenamiento, presa y depredador* 19
Figura 2.24: *Jerarquía de objetos del agente cazador* 20
Figura 2.25: *Escenario de la galería de tiro* 21
Figura 2.26: *Punto de vista del jugador en la galería de tiro* 21
Figura 2.27: *Recompensas acumuladas durante el entrenamiento de la galería de tiro* 22
Figura 3.1: *Escenario del juego de caza* 23
Figura 3.2: *Representación de las transiciones de estados* 24
Figura 3.3: *Entrenamiento del agente perseguidor* 24
Figura 3.4: *Recompensas acumuladas de los agentes del juego de caza* 25
Figura 3.5: *Zonas de entrenamiento con Wave Function Collapse* 26
Figura 3.6: *Recompensas acumuladas del agente que patrulla* 27
Figura 3.7: *Comparativa de recompensas acumuladas del agente que persigue* 27
Figura 3.8: *Comparativa de recompensas acumuladas del agente que dispara* 28

Índice de tablas

Figura 3.1: *Tiempos de entrenamiento* 30
Figura 6.1: *Resumen de tipos* 33

Capítulo 1 Introducción

En el mundo de los videojuegos, hoy en día existen numerosos títulos que utilizan mapas generados aleatoriamente. Un género que se aprovecha muy bien de estas técnicas de generación es el Roguelike, en el que en cada partida se genera un entorno totalmente diferente al anterior, esto incluye tanto al mapa, sus elementos y los personajes no jugador (PNJ) que lo habitan.

La creación de estos niveles puede llegar a ser todo un desafío. Sobre todo si se aspira a una cantidad potencialmente infinita de los mismos, contando al mismo tiempo con que la IA de los PNJ pueda adaptarse a todas las posibles variaciones, es importante tener en cuenta estos dos aspectos: la generación del mapa y el comportamiento de los PNJ.

Por un lado, la generación de mapas requiere de alguna técnica capaz de crear numerosas variaciones sobre el mismo concepto, y es aquí donde entra el algoritmo de Wave Function Collapse [3]. Es un algoritmo que ha ganado cierta popularidad, viéndose en juegos como Townscaper [5], por su simplicidad y efectividad a la hora de conseguir resultados comparables a los originados por redes neuronales convolucionales para este tipo de casos, pero con un rendimiento mayor.

Por otro lado, es posible establecer una serie de reglas manualmente para determinar el comportamiento de los PNJ que habiten el entorno, pero considerando que dichas reglas sean suficientes para que sepan adaptarse a todas las variaciones generadas . Es por ello por lo que resulta interesante plantear la opción de que la IA esté controlada por una red neuronal. Durante los últimos años, la popularidad de las redes neuronales ha ido exponencialmente en aumento, aplicándose a campos tan diversos como la literatura, la medicina, o la música, entre otros. Y, por supuesto, los videojuegos no son una excepción. Cabe destacar su versatilidad a la hora de enfrentarse a escenarios hasta entonces desconocidos según el entrenamiento que han recibido. En Unity existe un kit de herramientas para trabajar con redes neuronales, llamado ML-Agents [2].

Es por ello que el objetivo de este Trabajo de Fin de Grado propone el desarrollo de un sistema capaz de implementar ambas partes, la generación de mapas mediante Wave Function Collapse y el entrenamiento de PNJ o agentes inteligentes con ML-Agents en los escenarios generados. De este modo, se podría crear un nivel jugable en el que los agentes se enfrentaran entre sí o contra el jugador en una actividad competitiva dentro de este entorno aleatorio.

Específicamente, los objetivos que propone este Trabajo de Fin de Grado son:

- Comprender en mayor profundidad el algoritmo de Wave Function Collapse y sus posibles aplicaciones en el desarrollo de videojuegos.
- Entender el funcionamiento interno del kit de herramientas de ML-Agents para aprender cómo funciona, y ser capaces de entrenar agentes inteligentes con comportamientos complejos.
- Integrar ambas partes en un solo sistema para generar escenarios aleatorios y entrenar agentes para que sean capaces de adaptarse a cualquier entorno generado aleatoriamente.

1.1 Antecedentes

Como se comentó anteriormente, resulta interesante proponer la combinación de ambas partes para favorecer el desarrollo de videojuegos. Hasta donde sabemos, no existe ningún estudio anterior que investigue conjuntamente estas tecnologías, pero sí otras análogas.

Como se muestra en la investigación “Open-Ended Learning Leads to Generally Capable Agents” [6], es posible desarrollar agentes inteligentes mediante entrenamiento por refuerzo que sean capaces de llevar a cabo múltiples tareas, con un comportamiento generalizado para adaptarse a toda clase de desafíos. En su caso, el escenario se genera con una topología estática, pero con ligeras variaciones y objetos dinámicos con simulación de físicas. Los jugadores, controlados por agentes, están entrenados con redes neuronales mediante el aprendizaje por refuerzo profundo utilizando el algoritmo V-MPO [7], una variación del MPO (Maximum a Posteriori Policy).

Por separado, tanto Wave Function Collapse como ML-Agents sí han sido ámbito de estudio a lo largo de los últimos años. Las investigaciones más comunes sobre Wave Function Collapse tratan de traducir el algoritmo a nuevos lenguajes o plataformas, o ampliar sus características. Por ejemplo, una variación de Wave Function Collapse basado en grafos (“Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm”) [8], que demuestra que el algoritmo como método de generación de texturas se puede extender a formas no basadas en mallas; o la posibilidad de determinar restricciones de diseño que deba cumplir en “Enhancing wave function collapse with design-level constraints” [9].

Por otro lado, ML-Agents lleva en desarrollo múltiples años con infinidad de proyectos basados en esta tecnología a prácticamente todos los niveles de complejidad. Desde simples introducciones al kit de herramientas, hasta ejemplos más complejos, como “Tuning Proximal Policy Optimization Algorithm in Maze Solving with ML-Agents” [10], que busca la configuración idónea de los parámetros del algoritmo PPO (Proximal Policy Optimization) [11] para la resolución de rutas en laberintos. Esta investigación pone en evidencia la dificultad que supone encontrar la forma idónea de entrenar a un agente para que cumpla una tarea determinada usando este kit de herramientas.

Se hará uso del curso introductorio “ML-Agents: Hummingbirds” [12] propio de Unity para comprender mejor todas las características que se ofrecen dentro de ML Agents, para ponerlas en práctica en este proyecto.

Capítulo 2 Desarrollo del proyecto

En el capítulo anterior se ha realizado la introducción al proyecto, su contexto y su propósito. En este apartado se hablará de las tecnologías en las que se basa.

2.1 Wave Function Collapse

Wave Function Collapse es un algoritmo de generación de contenidos, popularmente utilizado en el ámbito de los videojuegos para la creación de mapas, pero muy útil además en otros aspectos, como la generación de imágenes, texturas o incluso música. En la figura 2.1 puede verse un escenario en tres dimensiones construido con Wave Function Collapse.

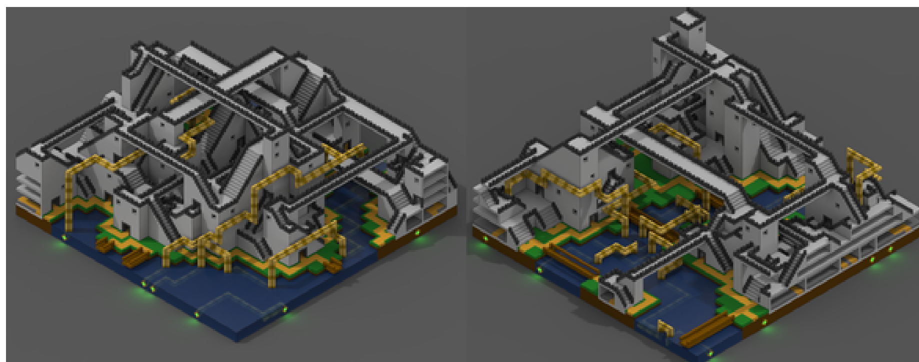


Figura 2.1*: Ejemplo de mapa 3D generado con Wave Function Collapse [3]*

Su funcionamiento es muy simple, parecido al de un sudoku. Se divide el entorno en casillas, cada una con una serie de valores posibles, se elige una aleatoriamente y se le asigna un posible valor. Esta modificación se propaga a sus vecinos aplicando una serie de reglas establecidas que determinan qué tipo de casillas pueden tener una relación de vecindad.

Con esta técnica es posible generar una casilla y asegurarse de que las contiguas son compatibles. Por ejemplo, si se quisiera generar el mapa de un bosque, los posibles elementos que podrían ir en cada casilla serían césped, un árbol, rocas, la orilla de un río, etc, y las reglas establecerían qué elementos pueden aparecer en casillas vecinas.

Hay dos maneras posibles de establecer las reglas, ya sea manualmente, determinando uno por uno qué elementos pueden ir juntos, o de manera automática. Esta última es un poco más compleja, pues se le pasa al algoritmo una muestra, pudiendo ser en el ejemplo anterior una parte muy pequeña del bosque. El algoritmo reconocerá qué elementos son contiguos y establecerá las reglas en función a ello. Hay que considerar que esta forma automatiza el proceso de creación de reglas, facilitando el trabajo al usuario, pero los resultados podrían no ser exactos a los deseados si la muestra otorgada no contiene las combinaciones necesarias. La figura 2.2 muestra cómo el algoritmo lee los patrones de la muestra y los aplica en la imagen generada, incluso llegando a rotarlos.

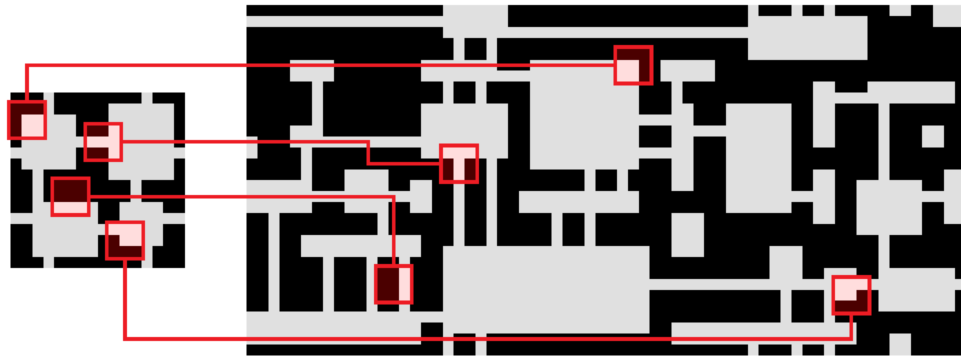


Figura 2.2: *Patrones de vecindad automáticos [3]*

Existen numerosas implementaciones del algoritmo, la mayoría como variaciones en el rendimiento dependiendo de la plataforma o el objetivo de esta. A su vez, J. Parker ha desarrollado una implementación [4] de código abierto en Unity para trabajar con este algoritmo desde el motor gráfico. En la figura 2.3 se puede observar un mapa generado con los assets de ejemplo que ofrece el proyecto de Wave Function Collapse; arriba la muestra, y abajo el resultado.

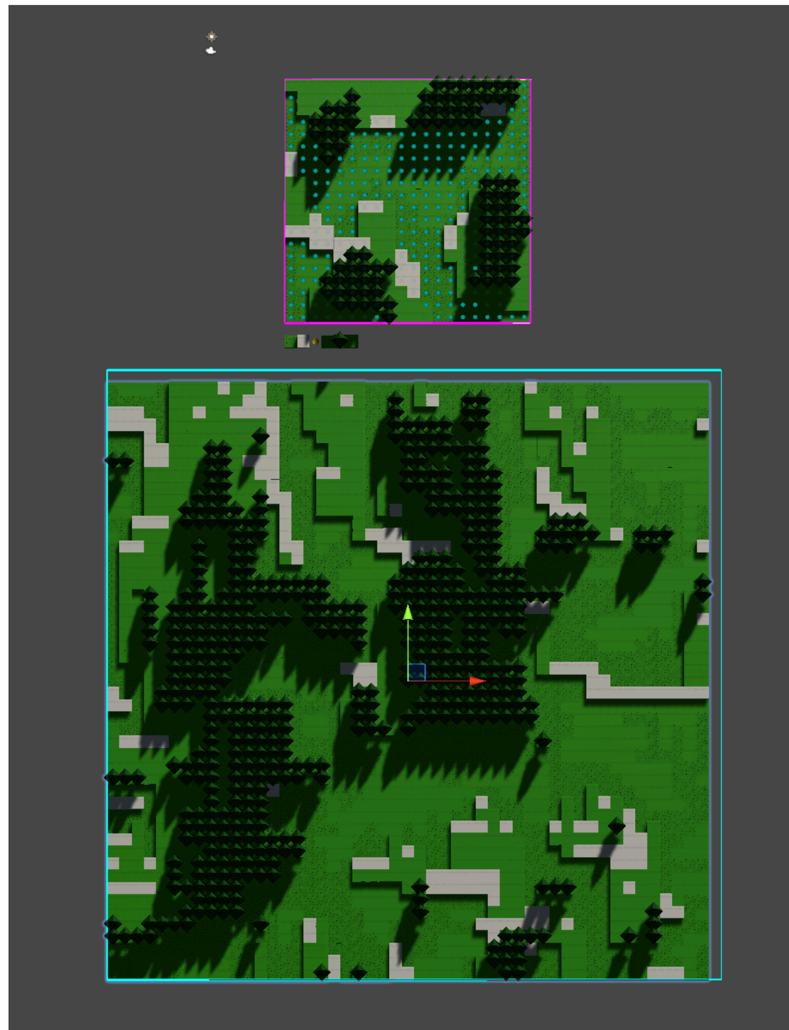


Figura 2.3: *Ejemplo generado con assets del proyecto base*

2.1.1 Desarrollo de los tipos de casilla

En un primer momento, se crearon solamente dos tipos de casillas, que representan suelo y pared. De esta manera, se podían construir escenarios sencillos en forma de laberintos para que los agentes pudieran entrenarse en tareas de búsqueda de objetivos y en determinar el recorrido hasta ellos, evitando posibles obstáculos. Las figuras 2.4 y 2.5 muestran estos dos tipos de celdas.

La casilla de tipo suelo o "Floor" se trata simplemente de un prefab en forma de cubo sin ninguna funcionalidad adicional, salvo por el componente Collider que posee, que evita que ciertos elementos del escenario puedan caer si se vieran afectados por la gravedad.

Figura 2.4: *Modelo de celda "Floor"*

El tipo de casilla pared o "Wall" es muy parecida a la anterior, pero dobla su tamaño en un eje para bloquear la visión de los agentes, y añade también un componente Rigidbody para detectar la colisión de los mismos en el caso de que llegara a ocurrir. Se ha activado la opción "Is Kinematic" para evitar que su posición se pueda ver afectada por dichas colisiones.

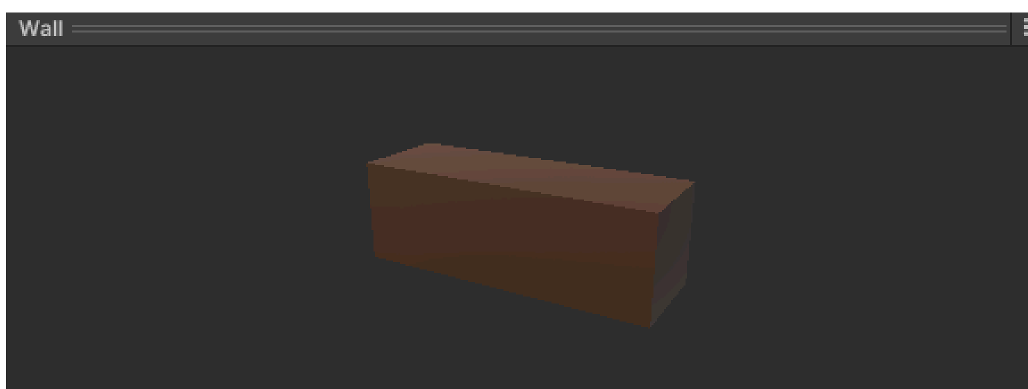


Figura 2.5: *Modelo de celda "Wall"*

De este modo, se pueden generar mapas sencillos con estos dos tipos de casillas. Añadiéndolas al canvas proporcionado por el proyecto de Wave Function Collapse, se

creará un pequeño ejemplo que represente un laberinto. Se puede variar la relación de casillas de muro respecto a las de suelo para aumentar o disminuir la densidad del mapa generado. En las figuras 2.6 y 2.7 se puede observar el cuadro pequeño a la izquierda, denominado canvas, que le sirve al algoritmo para determinar los patrones de adyacencia.

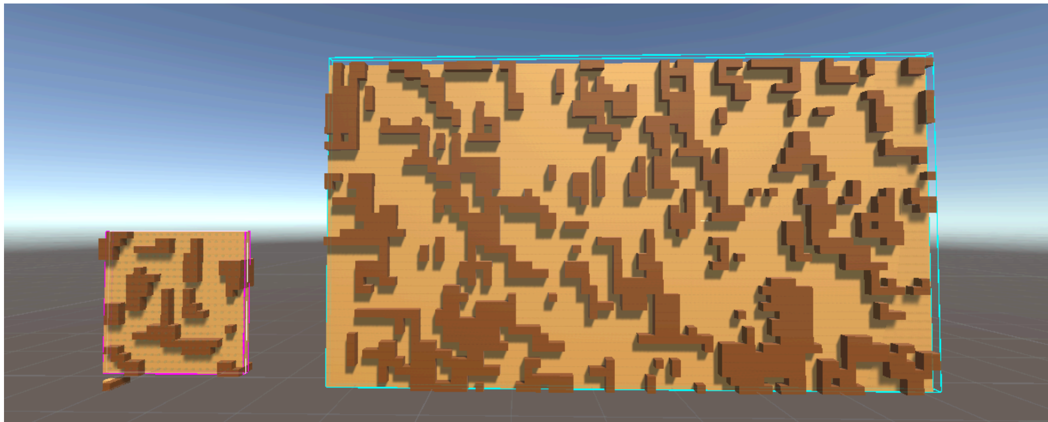


Figura 2.6: *Modelo generado con WFC, mayor densidad de muros*

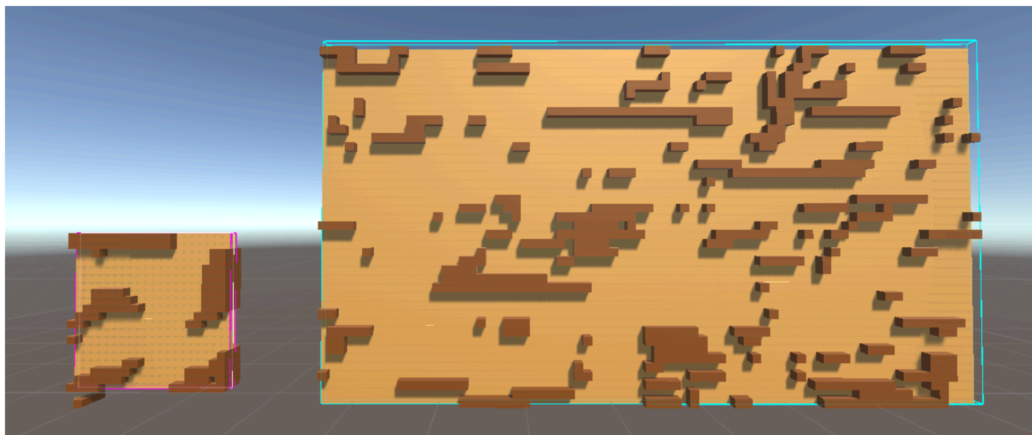


Figura 2.7: *Modelo generado con WFC, menor densidad de muros*

Si se compara el mapa generado con el modelo proporcionado en esta última figura, se puede apreciar que los patrones de muros no son del todo parecidos, notándose aún más la diferencia si se observa la Figura 2.3. Esto se debe a que el algoritmo funciona peor cuantos menos tipos de casilla disponga para generar el mapa, siendo dos en este ejemplo, y seis en la Figura 2.3.

Por último, se creó un nuevo tipo de casilla para aumentar la variedad de los mapas generados y dar más posibilidades a los agentes en las actividades que lleven a cabo. Esta es el conjunto de hierba o “Grass”, observable en la figura 2.8, y permite a determinados agentes “ocultar su presencia” frente a otros. Es como el prefab “Wall”, salvo que transparente, para simular el conjunto de hierba. Los agentes pueden ocultarse en ella y se ha activado la opción “Is Trigger” de su Collider para saber en qué momento

un agente entra en una casilla de este tipo. El funcionamiento de este sistema es muy sencillo, una vez un agente se encuentra dentro de esta casilla, se cambia la capa donde se encuentra a una nueva llamada “Void”, que solo puede interactuar con elementos de esta misma, impidiendo que los sensores de otros agentes puedan detectarlo.

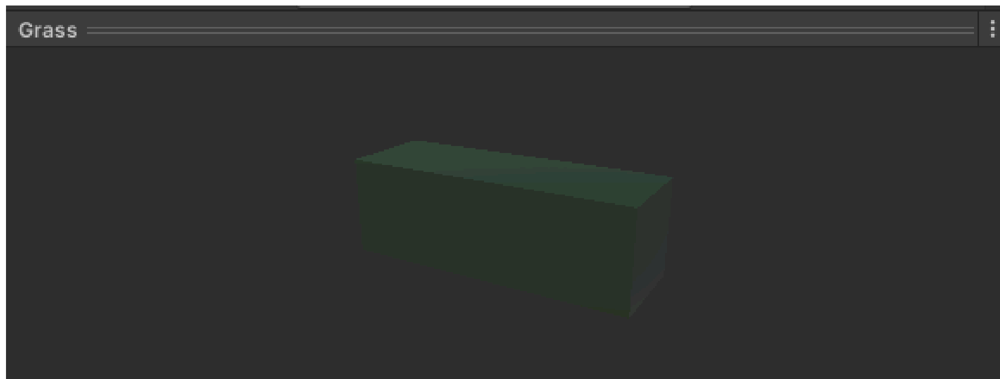


Figura 2.8: *Modelo de celda “Grass”*

Como se puede apreciar en la figura 2.9, los objetos que pertenezcan a la capa void solo pueden colisionar con otros de esta misma. Esto impide que los sensores de otros agentes puedan detectarlos, dado que funcionan mediante colisiones.

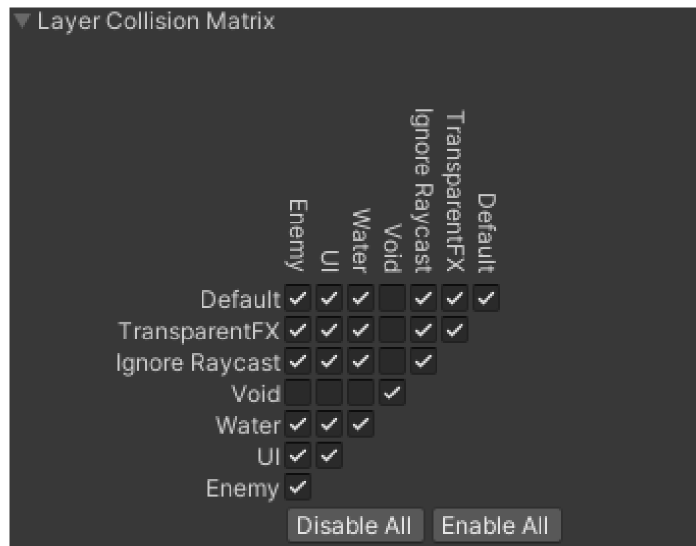


Figura 2.9: *Matriz de capas de colisión*

En la figura 2.10 se puede ver a la derecha, la muestra de donde se sacan los patrones, y a la izquierda, el resultado generado con los tres tipos de casillas desarrollados para este proyecto.

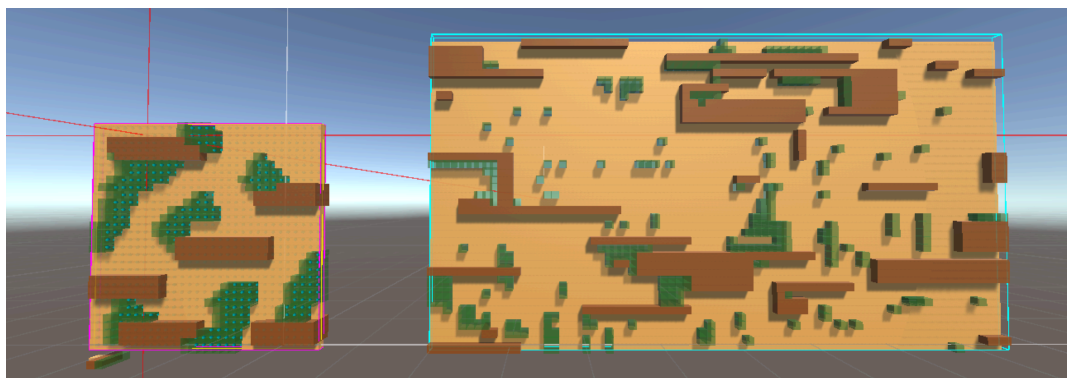


Figura 2.10: *Modelo generado con WFC con tres tipos de casillas*

2.1.2 Guardado y carga de mapas

El proyecto base de Wave Function Collapse permite generar mapas aleatorios en cualquier momento, incluso es posible repetir un resultado si se mantiene el valor de su semilla. Sin embargo, sería interesante encontrar una forma de guardar la información de estos mapas y poder cargarlos en el momento deseado sin tener que depender de la ejecución del algoritmo de nuevo. Por ello, se ha creado un sistema que lee qué tipo de casilla existe en cada posición de la matriz que representa el mapa, y se guarda el nombre de su etiqueta en una matriz de strings en un archivo JSON. Se ha elegido este formato porque es fácilmente exportable tanto dentro como fuera del proyecto si así se deseara.

El guardado se basa en un conjunto de scripts, uno de ellos simplemente aporta visualmente un botón en el inspector de Unity, que se añade como componente al Overlap del que se quiera almacenar su información, y se llama al método SaveData() de la clase SaveTiledMap. Este lee el tamaño del grid, el ancho y la profundidad para construir la matriz de strings, y luego recorre todas las casillas almacenando el nombre de la etiqueta en cada una, pudiendo ser “Floor”, “Wall” o “Grass”. Una vez hecho esto, se utiliza una clase ayudante, JSONHelper, que a través de otra envolvente (Wrapper) da el formato correspondiente a la información del mapa para pasársela a la función ToJson() de JsonUtility, quien se encarga de hacer la traducción a un string en formato JSON. Por último, se guarda este string en un documento ubicado en la carpeta Resources del proyecto.

Por otro lado, la carga de mapas es equivalente, pero el proceso contrario. Al igual que con el guardado, se añade un botón en el inspector del Overlap donde se quiera guardar la información, que llama a la función LoadData() de la clase LoadTiledMap. Esta se encarga de leer el fichero JSON del mapa deseado y se lo pasa a la clase JSONHelper para que haga la conversión entre ese formato y la información de las variables de tamaño de grid, ancho, profundidad y la matriz de strings con las etiquetas de las casillas. Para generar el mapa, se recorre cada posición de la matriz, se lee qué etiqueta tiene y se instancia el prefab correspondiente para dicha posición.

En la figura 2.11 se muestran los botones en el inspector que permiten la carga y el guardado de los mapas.

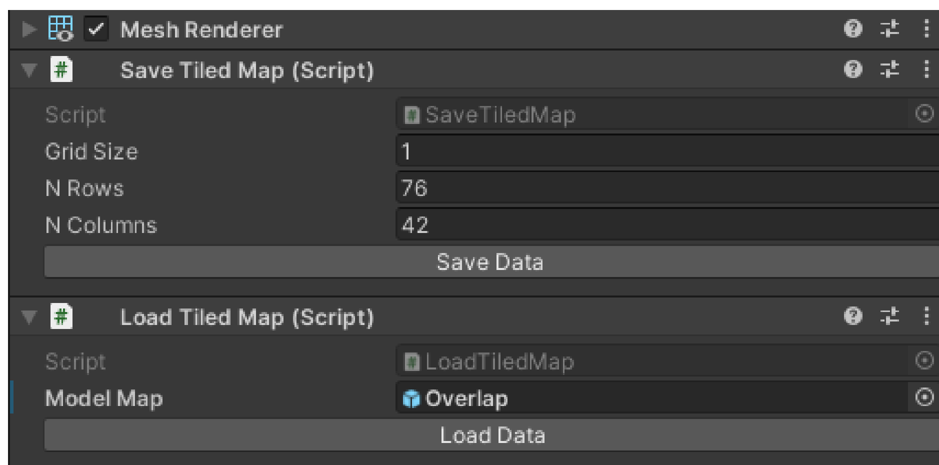


Figura 2.11: *Botones en el inspector para el guardado y carga de mapas*

2.2 ML-Agents

ML-Agents es un kit de herramientas propio de Unity para el desarrollo de entornos de entrenamiento de agentes inteligentes mediante Machine Learning. Se trata de un proyecto de código abierto que, basándose en PyTorch [13], ofrece diversos algoritmos para entrenar a estos agentes en juegos y simulaciones. Estos agentes pueden tener múltiples propósitos, entre los que están el control del comportamiento de PNJ. Además, se pueden usar diferentes tipos de entrenamiento, como el aprendizaje por refuerzo, por imitación, neuro evolución u otros. En el caso de este proyecto, se centrará la atención en el aprendizaje por refuerzo.

2.2.1 El aprendizaje por refuerzo

Es un tipo de proceso de Machine Learning que se centra en la toma de decisiones de agentes autónomos. Estos agentes son cualquier sistema capaz de tomar decisiones y actuar en respuesta a su entorno. En el caso de ML-Agents, el sistema latente es una red neuronal. En el entrenamiento por refuerzo, los agentes aprenden a realizar una tarea a base de prueba y error.

Esencialmente consiste en la relación entre el agente, el entorno y el objetivo, generalmente conocida como el proceso de decisión de Markov (MDP): el agente recibe información sobre el entorno y usa dicha información para determinar su siguiente acción. Esta acción cambia el estado del entorno y tiene asociada una cuantía como una recompensa o una penalización, dependiendo de si el agente se comporta como se espera o no en función del objetivo establecido. Esta recompensa o penalización influye en el entrenamiento de la red neuronal, que ajustará sus ponderaciones para favorecer que el agente repita esa acción cuando se vuelva a encontrar en un estado similar. Este proceso se repite un número determinado de iteraciones, de forma que los agentes acumulen experiencia en una amplia variedad de estados. La figura 2.12 explica visualmente esta relación.

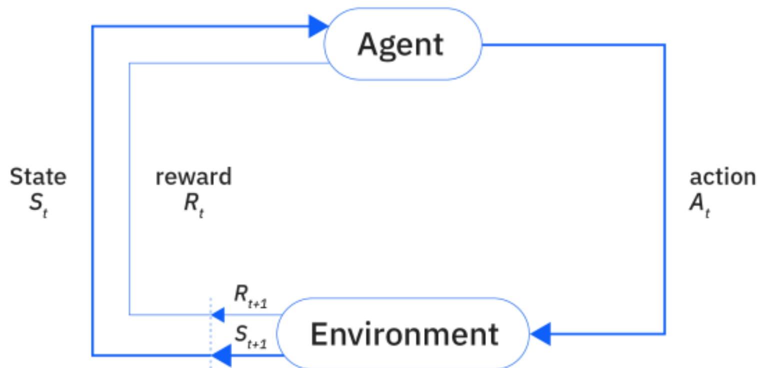


Figura 2.12: *Proceso de decisión Markov [14]*

El número de pasos que da un agente entre el momento en el que empieza una actividad hasta que la completa, o en su defecto, que falle su objetivo, se conoce como episodio.

Más allá del conjunto agente/entorno/objetivo, hay cuatro subelementos que caracterizan el aprendizaje por refuerzo.

- Política: define el comportamiento del agente al mapear los estados con determinadas acciones que debería tomar. Puede tener la forma rudimentaria de una función o procesos computacionales más complejos.
- Señales de recompensa: cada acción tomada puede añadir o sustraer parte de la recompensa, y el agente actuará con el único objetivo de maximizar dicha recompensa.
- Función de valor: la diferencia con las señales de recompensa reside en que las primeras se centran en recompensas a corto plazo, mientras que estas últimas a largo plazo. La función de valor se define como de deseabilidad de un estado para todos los posibles estados que puedan seguirlo. Por ejemplo, puede que una determinada acción maximice la recompensa, pero dicha acción llevaría a un comportamiento poco deseado por parte del agente en el futuro, disminuyendo su deseabilidad.
- Modelo: los modelos de predicción son un elemento opcional que permite a los agentes determinar posibles cursos de acción.

2.2.2 Componentes clave de ML-Agents

Como se nombró antes, ML-Agents dispone de numerosos tipos de entrenamientos, pero en este proyecto se hará hincapié en el aprendizaje por refuerzo. Con este kit de herramientas es posible entrenar a los agentes para realizar tareas, y su efectividad y exactitud dependerán de cómo y en qué situaciones se aporta la recompensa anteriormente mencionada, las observaciones percibidas por el agente y las acciones que pueda realizar.

Las observaciones son todos aquellos datos que recibe el agente y puedan ser

relevantes para llevar a cabo su tarea, siendo numéricos o visuales. Por ejemplo, un agente entrenado para llevar un objetivo a una meta debería tener como observaciones la posición de ambos, como mínimo. Se pueden añadir más observaciones si se desea un comportamiento más complejo (que sea capaz de percibir obstáculos), pero es importante tener en cuenta la relevancia de estas, pues pueden influir negativamente en los resultados. En cuanto a las acciones, son todas aquellas que pueda realizar el agente. En el ejemplo anterior, podría ser simplemente moverse en el plano XZ para ir de un punto a otro.

Una vez definidas estas tres entidades fundamentales para el entrenamiento de una tarea, se debe hablar de los componentes clave de alto nivel del kit de herramientas de ML-Agents.

- Entorno de aprendizaje: contiene la escena de Unity, los personajes de juego y otros elementos. Es donde el agente entrenará y su disposición dependerá de su tipo.
- API de bajo nivel de Python: contiene una interfaz para interactuar y controlar el entorno. Esta API es independiente de Unity, pero se comunica con este a través del Comunicador.
- Comunicador externo: elemento que se encuentra dentro del entorno y lo conecta con la API.
- Entrenadores Python: contienen todos los algoritmos de ML que posibilitan el entrenamiento de los agentes. Estos algoritmos están implementados en Python.
- Wrappers: los Wrappers [14] son una forma común con la que los investigadores interactúan con los entornos de aprendizaje sin tener que modificar el código directamente. ML-Agents proporciona dos diferentes: Gym y PettingZoo, para modificar los entornos y obtener simulaciones con múltiples agentes, respectivamente.

En la figura 2.13 se puede observar cómo se establece la comunicación entre la API de Python, el entrenador y el entorno de aprendizaje.

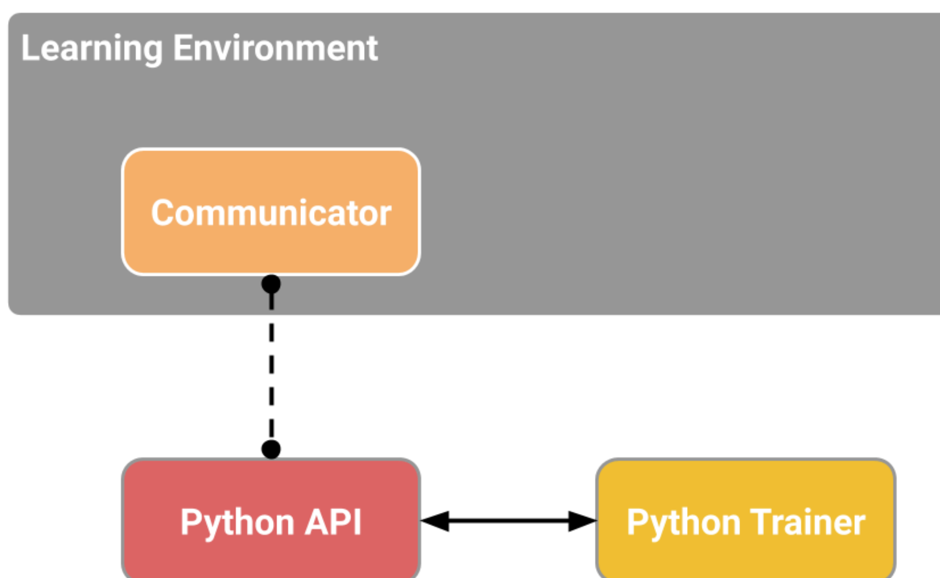


Figura 2.13: *Componentes clave de ML-Agents [14]*

2.2.3 Componentes del entorno de entrenamiento

El entorno contiene dos componentes de Unity:

- Agentes: asociados a un GameObject de Unity, y maneja la recolección de observaciones, las acciones y la asignación de las recompensas. Cada agente está vinculado a un comportamiento.
- Comportamiento: define una serie de características del agente, como el número de acciones que puede llevar a cabo, o las observaciones que recibe, y está identificado con un nombre único. Puede ser de tres tipos:
  - Aprendizaje: aún está por entrenar.
  - Heurístico: definido manualmente por una serie de reglas implementadas en el código.
  - Inferencia: un comportamiento que ya ha sido entrenado a través de la red neuronal.

La figura 2.14 muestra de forma expandida el entorno de entrenamiento, con varios agentes que comparten o no el mismo comportamiento.

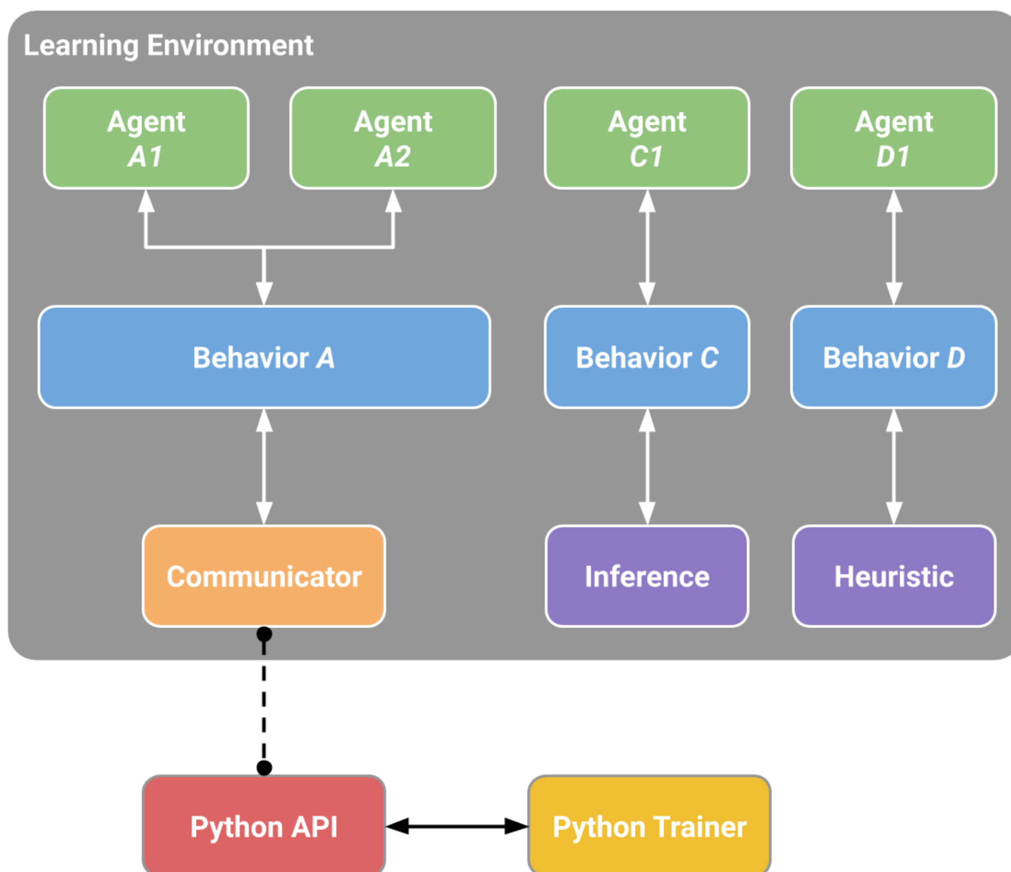


Figura 2.14: *Componentes del entorno de entrenamiento [14]*

2.2.4 Proximal Policy Optimization

ML-Agents ofrece la implementación de dos algoritmos de aprendizaje por refuerzo, siendo PPO o Proximal Policy Optimization el utilizado para el desarrollo del comportamiento de los agentes de este proyecto. Es un método basado en políticas, que determina directamente la mejor acción dado un estado concreto, a diferencia de los basados en valores, que necesitan aprender primero los pares estado/acción.

El funcionamiento de PPO consiste en generar tuplas de estado, acción y recompensa para cada episodio. Luego se define una trayectoria como un conjunto de estas tuplas al finalizar múltiples episodios, y la información recabada se utiliza retroactivamente para actualizar la política.

Un componente vital es la estimación del rendimiento de una acción respecto al comportamiento por defecto de la política. Si el resultado real difiere significativamente del estimado, significa que se requiere actualizar los pesos en la red neuronal.

En esta parte del proyecto se explicará paso a paso cada fase del desarrollo, tanto de Wave Function Collapse como de ML-Agents.

Es importante remarcar antes de comenzar la explicación del desarrollo la diferencia de los vectores de acciones discretos y continuos. Estos vectores determinan cuántas acciones pueden llevar a cabo los agentes y qué valores pueden tomar; siendo 0 ó 1 para las discretas, y entre -1 y 1 para las continuas.

En los ejemplos se han utilizado acciones continuas para el movimiento de los agentes por el hecho de que resulta más natural al poder tomar cualquier valor dentro del rango. La excepción es en el entorno de presa contra depredador a modo de prueba. Aparte, la acción de disparar del agente cazador también lo es por conveniencia, actuando como una variable booleana.

Cada ejemplo desarrollado se basa en el anterior con el objetivo de implementar una demostración jugable más compleja al final, empezando con un simple movimiento hacia un objetivo estático, y finalizando con un entorno con múltiples agentes con diferentes comportamientos, donde el jugador puede interpretar cualquiera de los papeles. La unidad para medir el rendimiento es el número de pasos que da el agente hasta cumplir con su actividad en cada episodio, intentando minimizarlos siempre y cuando no interfiera en su tarea.

Alcanzar el objetivo

El primer ejemplo desarrollado fue el de un agente que tuviera que alcanzar a un objetivo estático. Se trata de un escenario muy sencillo del que parten los demás para conocer el funcionamiento básico de ML-Agents, representado en la figura 2.15.

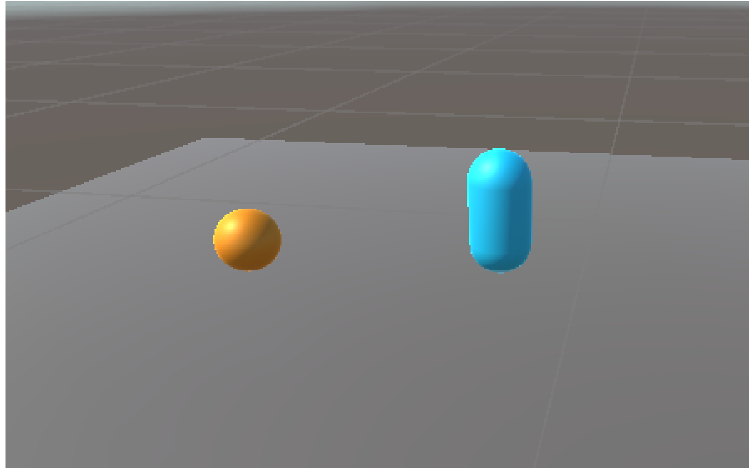


Figura 2.15: *Escenario de alcanzar al objetivo*

El objetivo es una esfera amarilla con un Collider que actúa como trigger para cuando el agente entre en contacto con ella. El agente es una cápsula azul, y en sus parámetros de comportamiento se puede ver que su espacio de observaciones es seis, tres por su posición (ejes X, Y y Z) y la del objetivo (ejes X, Y y Z). Al mismo tiempo, solo dispone de dos acciones: moverse en el eje X y en el eje Z. Esta información es apreciable en los parámetros de comportamiento del agente, vistos en la figura 2.16. En este caso, la única manera tiene el agente para conseguir una recompensa es alcanzar su objetivo, aumentándola en 10 puntos; y la pierde si llega a colisionar contra un muro o al salirse de los límites del mapa, reduciéndose en 7.5 y 10, respectivamente. Además, se resta 0.01 en cada paso para incitar al agente a no tomar acciones innecesarias.

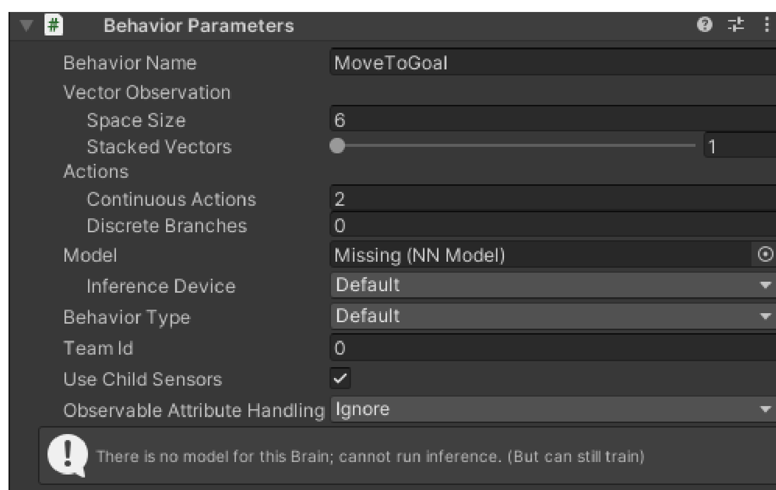


Figura 2.16: *Parámetros de comportamiento del primer agente*

Otra opción hubiera sido permitirle moverse en un solo eje, pero rotar en Y para poder cambiar de dirección. Como por ahora no es relevante hacia dónde apunta, se ha optado por esta forma, pero más adelante sí cobrará protagonismo.

El agente solo necesita recabar como observaciones su posición y la de su objetivo, y con esa información puede ir de un punto a otro con un número de pasos prácticamente mínimo; es decir, el agente suele dar un paso que le acerque a su objetivo. Es importante remarcar que este escenario es muy simple, solamente de prueba. Al añadir obstáculos o probarlo en mapas generados por WFC, el agente tiende a trazar una ruta en línea recta sin tener en cuenta que haya muros interrumpiéndola. Es por ello que en los siguientes ejemplos se ha tenido en cuenta este detalle.

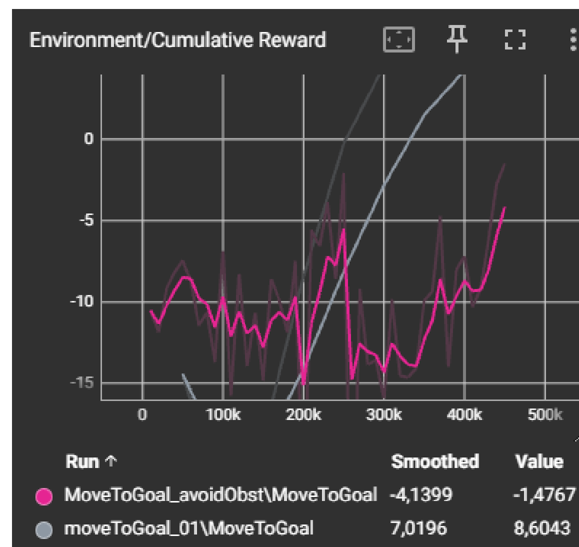


Figura 2.17: *Recompensas acumuladas del agente que se mueve hacia un objetivo*

La figura 2.17 muestra la obtención de la recompensa acumulada de este primer agente cuando se han añadido los obstáculos (rosado), en forma de tres muros colocados a cierta distancia. Se puede observar lo errática que es, aumentando y disminuyendo abruptamente la recompensa por las continuas colisiones contra los muros del escenario, pues el agente no dispone de las observaciones necesarias para poder esquivarlos. En comparativa, en el escenario sin obstáculos (gris) el aprendizaje es mucho más claro y rápido.

Presa contra depredador

Este escenario plantea el enfrentamiento de dos agentes diferentes: una presa y un depredador. Como sus nombres indican, uno debe escapar del otro. Sin embargo, ML-Agents solo permite el entrenamiento de un tipo de comportamiento al mismo tiempo. Partiendo de la base anterior, la presa aparecerá en un punto aleatorio del mapa y tratará de alcanzar un objetivo estático, mientras esquiva al depredador. Por otro lado, el

depredador también aparecerá en una posición aleatoria y perseguirá a la presa ignorando por completo al objetivo.

Se podría escoger el modelo anteriormente entrenado para la presa, pero debido a su sencillez y que no se comporta correctamente en escenarios más complejos, es necesario desarrollar más su entrenamiento. Esta vez se han añadido tres muros al escenario para probar la capacidad de los agentes para evitarlos, pero se debe buscar un método para que puedan detectarlos. No se les puede pasar la posición de cada muro como se hizo con la del objetivo, pues no es escalable en entornos grandes.

ML-Agents dispone de varios tipos de sensores que podrían ser la solución a este problema. El primero de ellos que se ha probado es el sensor de rayos de percepción. Consisten en una serie de rayos con origen en el agente, dirigidos hacia delante en forma de abanico y que pueden detectar aquellos objetos cuyas etiquetas coincidan con las que se definan como “Detectable Tags”; en este caso, las etiquetas que corresponden con el objetivo, los muros y los límites del mapa. En la figura 2.18 se puede apreciar la configuración del sensor, y en la figura 2.19, su representación en tres dimensiones.

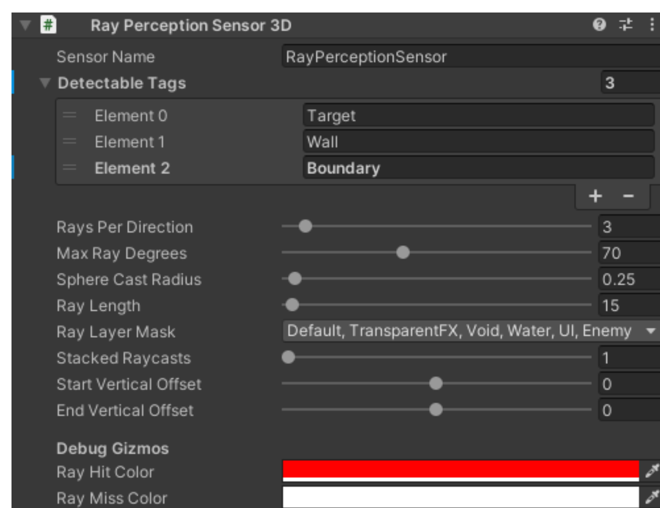


Figura 2.18: *Sensor de percepción de rayos del agente depredador*

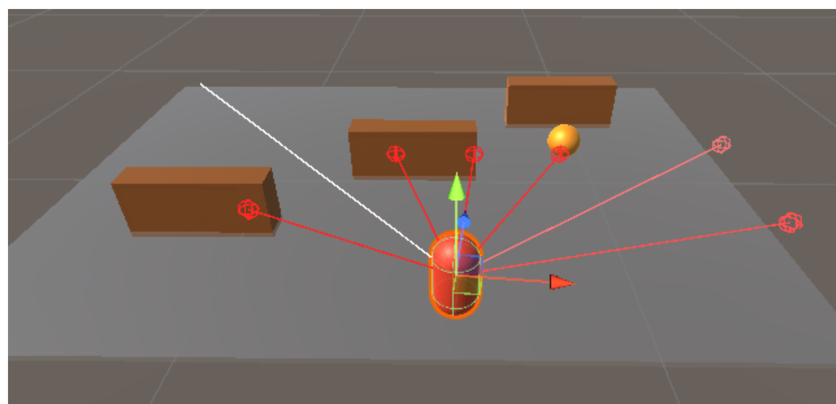


Figura 2.19: *Primera iteración del escenario presa contra depredador, mostrándose solamente el agente presa y su objetivo*

En este punto sí es relevante la orientación del agente respecto al eje Y, pues le permitiría girar sobre sí mismo para buscar su objetivo. Por otro lado, el número de observaciones no tiene por qué cambiar. Los rayos de percepción se añaden a las observaciones del agente sin tener que modificar su vector de observaciones manualmente. Teniendo en cuenta que ya sería capaz de detectar al objetivo sin pasarle en todo momento su posición, se podría eliminar este dato de las observaciones, pero solo lo percibiría cuando esté frente a él. Se ha decidido mantener este dato por ahora. Las recompensas se reparten de la misma manera que en el ejemplo anterior. Sin embargo, los resultados obtenidos no son tan favorables como se esperaban, pues el agente suele colisionar a menudo con los muros del escenario a pesar de recibir una penalización por ello, y a veces se mantiene a cierta distancia del objetivo sin llegar a alcanzarlo durante un tiempo.

Tras varias iteraciones intentando solucionar estos defectos en el comportamiento, se llegó a una versión que usa un sensor en forma de matriz de Colliders en forma de cubo alrededor del agente, con él en el centro. Funciona del mismo modo que el sensor de rayos comparando etiquetas, cada Collider de la matriz observa la etiqueta del objeto con el que colisiona y si está definida como "Detectable Tags". En la figura 2.20 se ve la configuración del sensor de matriz y en la 2.21 la representación en tres dimensiones del mismo.

Para el agente presa finalmente se ha optado por incrementar su recompensa en 0.005 puntos en cada paso que da, para fomentar su supervivencia, pero se resta 0.05 si se encuentra muy cerca del depredador. Se ha observado que de esta manera y junto al sensor de matriz, se consigue que el agente evite mejor tanto al depredador como a los muros. Además, se probó a cambiar el espacio de acciones a discreto para comprobar si este cambio en el tipo de movimiento influiría en su comportamiento. Esto solo provocó que el agente utilizara movimientos ortogonales y diagonales exclusivamente, sin influenciar significativamente en los resultados.

Posteriormente y debido a la ligera mejoría, se desarrolló el agente depredador a partir del de la presa, modificando el objetivo que debía alcanzar y sus observaciones. Cabe destacar que la exigencia de cómputo del sensor de matriz es notablemente superior que el de rayos, pues debe comprobar cada una de las casillas, provocando que aumente el tiempo de entrenamiento.

Para mantener un código de colores de ahora en adelante, la cápsula verde representará a la presa, y la roja al depredador. La figura 2.22 muestra el resultado final de este ejemplo.

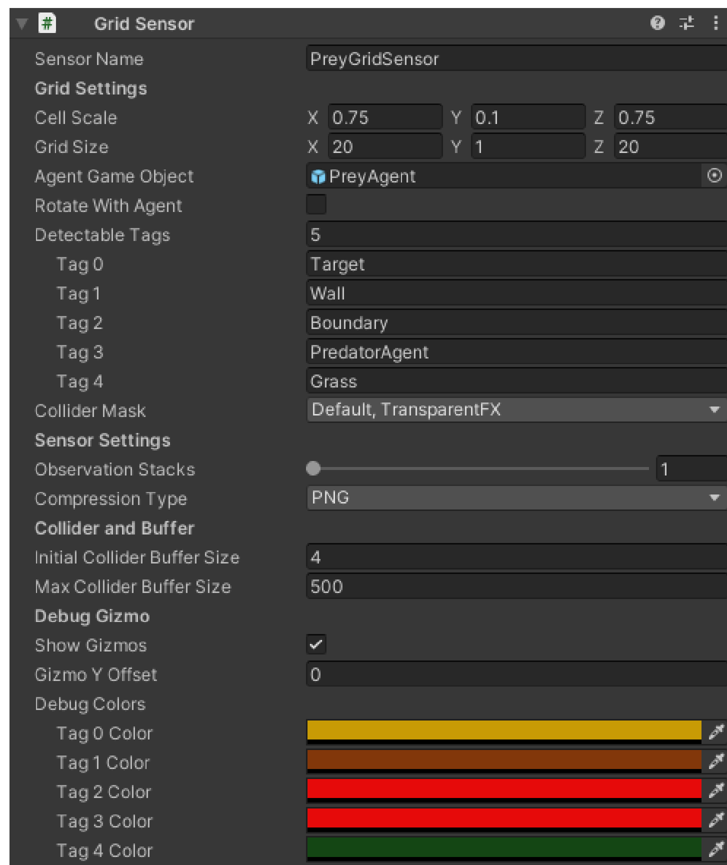


Figura 2.20: *Sensor de percepción de matriz del agente presa*

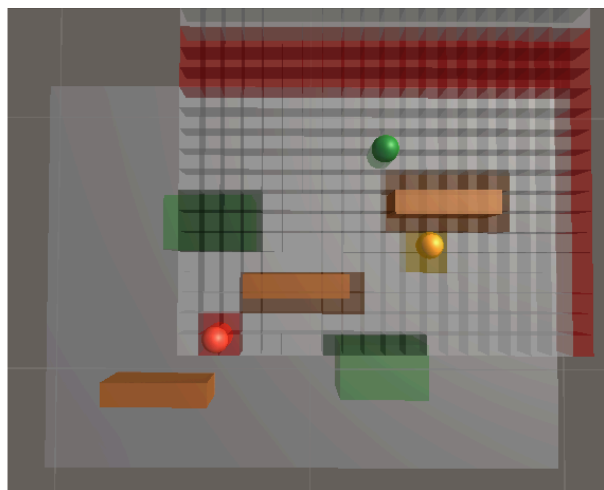


Figura 2.21: *Representación visual del sensor de percepción de matriz*

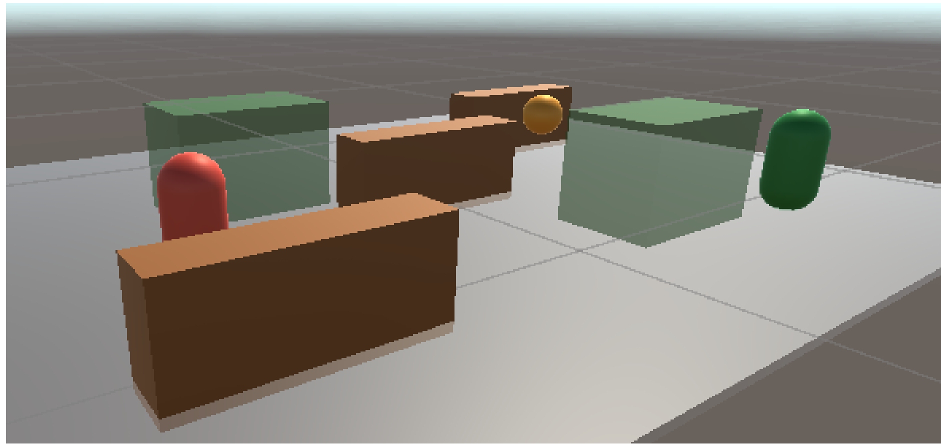


Figura 2.22: *Iteración final del escenario presa contra depredador*

A pesar de que las casillas de tipo “Grass” proporcionaban una ayuda al agente presa, no se percibió un uso intencionado de las mismas por su parte. Esto es principalmente debido a que el agente no tiene una forma clara de relacionar la acción de ocultarse con su supervivencia para alcanzar su objetivo. Se probó a premiar al agente al ocultarse, pero daba lugar a que el agente solo se dedicaba a entrar y salir de estas casilla, ignorando su objetivo principal.

Este resulta ser el ejemplo con la obtención de recompensas de ambos agentes más similares. Como se puede comprobar en la gráfica de la figura 2.23, ambas curvas crecen relativamente con la misma forma y estabilizándose aproximadamente en el mismo punto, incluso habiéndose entrenado por separado. Esto significa que el entrenamiento de cada uno evolucionó de manera similar, pues sus objetivos realmente son parecidos, y se ve reflejado en su comportamiento con los modelos ya entrenados, tanto uno como otro cumplen con su designación de manera efectiva, dejando la victoria a manos de quien sea colocado en una posición más favorable.

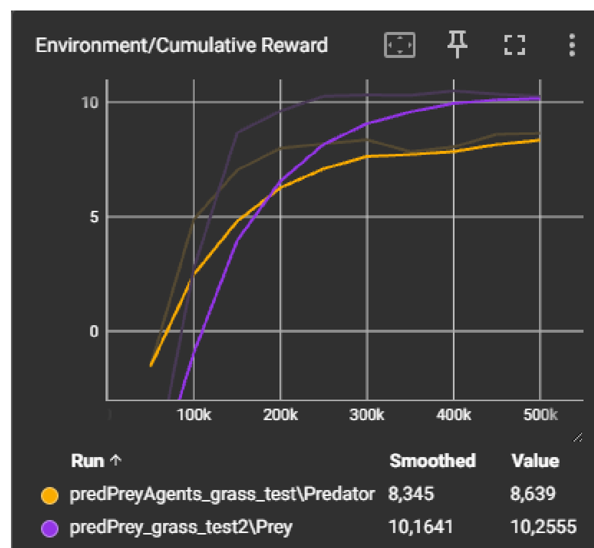


Figura 2.23: *Recompensas acumuladas durante el entrenamiento, presa y depredador*

Galería de tiro

Este ejemplo plantea un escenario donde un agente puede disparar proyectiles a una serie de objetivos estáticos, llamados “Enemies”, aumentando una puntuación única para ese agente. Se ha preparado el escenario para permitir la existencia de múltiples agentes simultáneos en una especie de juego competitivo, donde el jugador también puede participar. Para ello, se ha proporcionado una cámara y una interfaz gráfica a cada agente, desactivándose si están siendo controlados por el modelo entrenado. Además, se ha modificado la estructura de componentes del agente, añadiendo el sensor de rayos frontal y uno trasero para evitar que se salga de los límites del mapa si el agente va hacia detrás. También dispone de un punto que funciona como origen de cada disparo. La figura 2.24 muestra la jerarquía de objetos que usará el agente que dispara a partir de este momento.

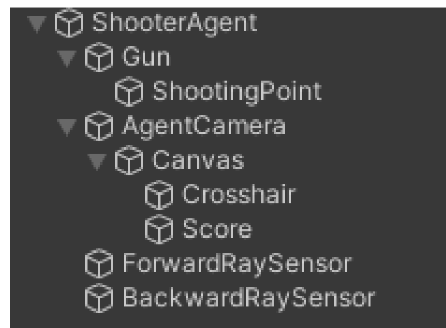


Figura 2.24: *Jerarquía de objetos del agente cazador*

Tras un periodo de investigación y documentación, se ha variado significativamente la forma de recolectar las observaciones por parte del agente. En los ejemplos anteriores, se pasaban todos los datos como observaciones directamente. Ahora, todas las observaciones se han normalizado con valores entre 0 y 1, y se ha ideado un sistema para detectar el enemigo más cercano al agente en cada momento, observándose el vector hacia el mismo y el ángulo de apuntado. También se observa la rotación del agente, si tiene un disparo disponible, si el enemigo al que apunta está en rango, si la trayectoria de disparo está bloqueada y cuál es la distancia al enemigo más cercano en relación a la distancia máxima posible dentro del escenario. Esto suma un total de 12 observaciones

En una primera implementación, para simplificar la parte del entrenamiento relativa al apuntado, no se le dio al agente la acción de disparar. Esta característica se añadió una vez se encontró una configuración óptima que le permitiera apuntar correctamente. Al ver que se obtenían buenos resultados, se desarrolló un disparo en forma de un *raycast*, restando 0.05 puntos de la recompensa si fallaban el tiro, y con tiempo de espera entre un disparo y otro. De esta manera, el agente obtiene dos vectores de acciones: uno continuo con 3 acciones para el movimiento en el eje XZ y la rotación en el eje Y; y otro discreto con dos acciones, disparar o no disparar.

Más adelante, se sustituyó el disparo en forma de *raycast* por un proyectil. Se trata simplemente de una esfera a forma de bala con un collider que actúa como trigger. Al

entrar en contacto con un enemigo, se activa la función OnTriggerEnter() de Enemy y se destruye, incrementando la puntuación del agente que efectuó el disparo.

Se desarrolló además una variante con obstáculos para comprobar si los agentes tendrían en cuenta si la trayectoria de disparo está bloqueada o no. En esta versión, el jugador competiría contra un agente, apareciendo en un lado del escenario, para eliminar el mayor número de enemigos posibles, como si de una galería de tiro se tratase, tal y como aparecen en las figuras 2.25 y 2.26.

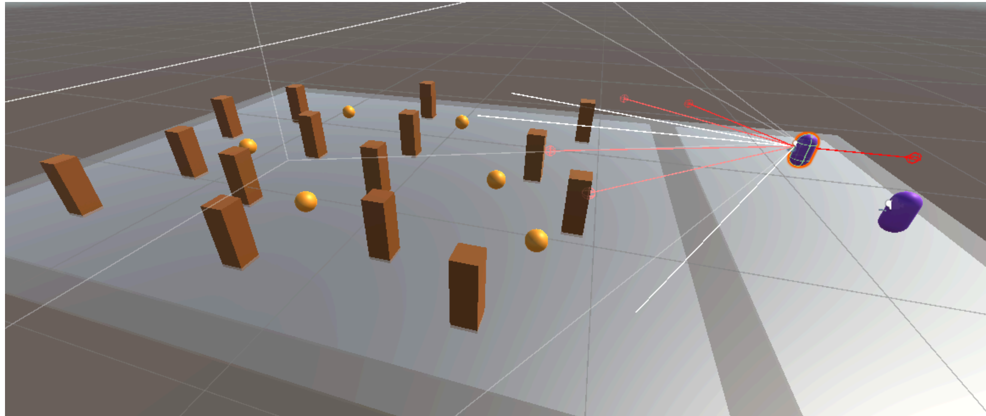


Figura 2.25: *Escenario de la galería de tiro*

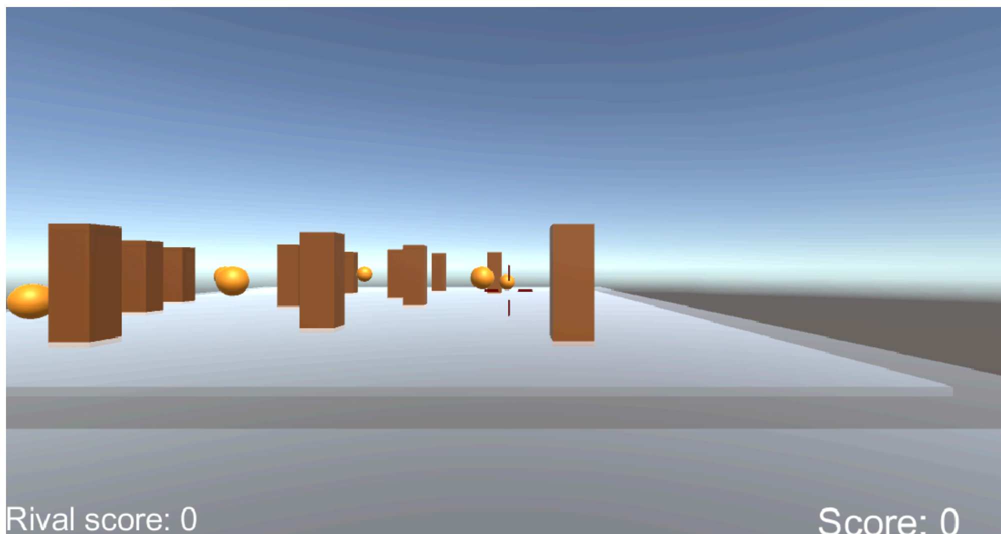


Figura 2.26: *Punto de vista del jugador en la galería de tiro*

En la figura 2.27 se puede observar, según la forma de la curva de recompensas acumuladas, que el agente podría parecer que estabiliza la obtención de la recompensa al finalizar el entrenamiento, pero existen varias desviaciones que pueden indicar lo contrario, tanto que siga aumentando como que disminuya. Una posible explicación podría ser que el agente falla muchos tiros, ya sea por un mal ángulo de apuntado o por la colisión de la bala con uno de los muros.

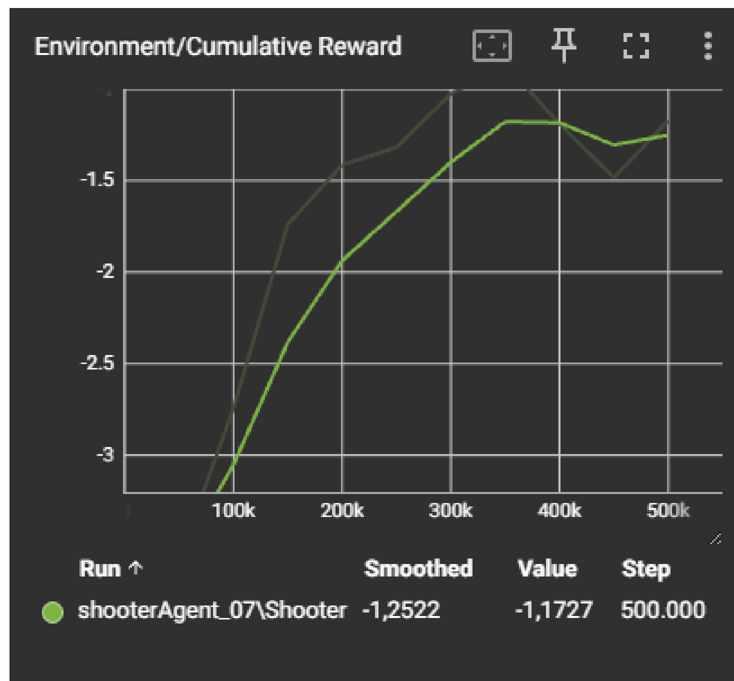


Figura 2.27: *Recompensas acumuladas durante el entrenamiento de la galería de tiro*

Capítulo 3 Resultados obtenidos

3.1 Juego de caza

Como desarrollo final y ejemplo que se ha basado sobre el resto de escenarios, se ha preparado un entorno que combina el entrenamiento de los agentes con un mapa generado con WFC para comprobar cómo se adaptan a la disposición aleatoria de sus elementos. La figura 3.1 muestra la vista cenital de un ejemplo. El conjunto de agentes lo forma el que vimos en el ejemplo anterior, que disparaba a objetivos estáticos, al que llamaremos cazador; y el enemigo, esta vez como cápsula roja, que hereda en parte el comportamiento del agente presa. El cazador tratará de disparar al enemigo siempre que esté en su rango, mientras que el enemigo intentará recolectar los objetivos que encuentre (esferas amarillas).

Figura 3.1: *Escenario del juego de caza*

Se ha planteado hacer más complejo el comportamiento del agente cazador, intentando que patrulle una zona, y cuando detecte a un enemigo, lo persiga hasta tenerlo dentro de su rango de tiro y luego le dispare. Esto puede complicar el entrenamiento, pues el agente debe tener en cuenta muchos factores a la vez, lo que le llevaría a un peor desempeño de su tarea. Es por ello que se ha optado por la implementación de ML-Agents con una máquina de estados finita. De esta manera, como se ve en la figura 3.2, se definen tres estados: patrulla, persecución y disparo; y se entrena un comportamiento más pequeño y sencillo para cada uno de los estados. Al transicionar de un estado a otro, se intercambia el agente actual con el correspondiente al nuevo estado. La variable de la que dependen las transiciones de estado es la distancia entre el agente y el enemigo, representada en la figura anterior por la línea roja.

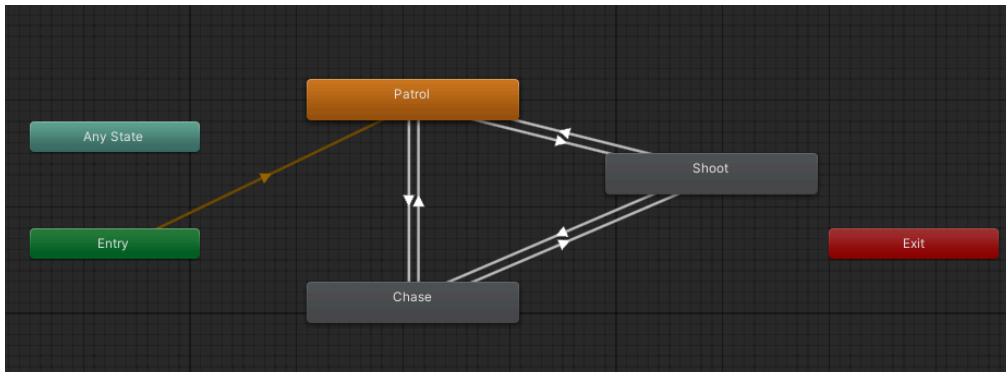


Figura 3.2: *Representación de las transiciones de estados*

3.1.1 Entrenamiento en escenario simple

En primer lugar, se ha entrenado a cada agente por separado en un escenario de prueba sin obstáculos para facilitar el ajuste de observaciones y recompensas. En la figura 3.3 se tiene un ejemplo del entrenamiento del agente perseguidor en este escenario. Más adelante se pasará a mapas generados con Wave Function Collapse.

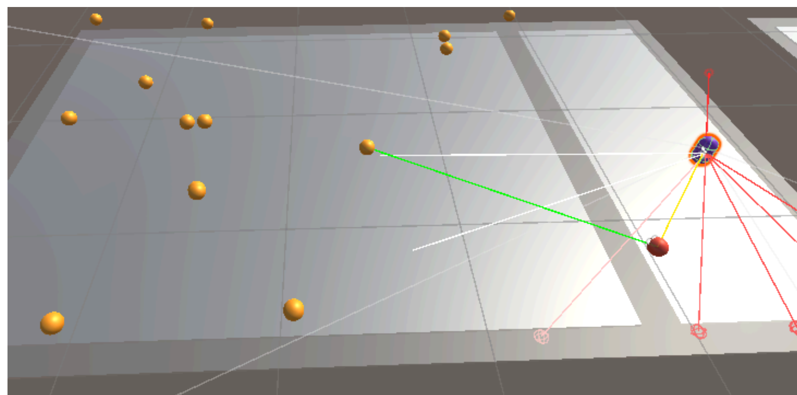


Figura 3.3: *Entrenamiento del agente perseguidor*

El agente que patrulla tiene un comportamiento muy sencillo, simplemente se encarga de recorrer la zona hasta que encuentra a un enemigo. Este agente observa su posición inicial de aparición, la distancia y el vector entre ambos, para ser capaz de mantenerse cerca de dicha zona, pues solo puede aumentar su recompensa mientras se encuentre a cierta distancia de la misma. Cuando se encuentre suficientemente cerca de un enemigo, se considera que lo ha detectado y se sustituye por el agente perseguidor.

Ahora, este agente tratará de acortar distancias con el enemigo hasta estar en rango de tiro mientras le apunta. Ya no se necesitan las observaciones relacionadas con la posición de aparición, pero se tiene en cuenta el ángulo hacia delante del agente respecto a la posición del enemigo y la distancia con él. Sus recompensas aumentarán si apunta correctamente a un enemigo y si se encuentra a menos de cierta distancia de él.

Una vez se ha acercado lo suficiente, se sustituye por el agente que dispara. Además de las observaciones anteriores, recibe si tiene disponible o no un disparo, si la trayectoria está libre de obstáculos y si sigue en rango de tiro. Este agente recibe recompensas si apunta correctamente a un enemigo y si acierta un disparo, y se le reducen en caso contrario.

En cuanto a las acciones de los agentes, se ha eliminado el movimiento en el eje X, limitándolo solo al eje Z. De la forma anterior, los agentes tendían a moverse hacia los límites del mapa hacia atrás, terminando prematuramente los episodios. Además, se le ha dado una velocidad frontal pasiva para obligarles a ir hacia delante la mayor parte del tiempo. Esto ha resultado en movimientos alrededor del mapa en forma de círculos en su mayoría para evitar colisionar con los límites del mapa.

La figura 3.4 muestra las curvas de recompensas acumuladas de los tres agentes; rosada para el que patrulla, gris para el que persigue y azul para el que dispara. El agente que patrulla estabiliza rápidamente las recompensas que gana. Esto significa que en ese punto la obtención de la recompensa es máxima con sus parámetros de comportamiento. Mientras, la pendiente de los otros dos agentes indica que sigue habiendo espacio de mejora. Con comportamientos más complejos, sería necesario aumentar el tiempo de entrenamiento; sin embargo, 500000 pasos deberían ser más que suficientes para estos casos.

Esto se puede deber a que el agente que patrulla solo necesita quedarse dando vueltas alrededor de una zona para aumentar su recompensa, mientras que los otros dos agentes tienen restricciones más exigentes. De igual forma, el desempeño de estos dos últimos es mejorable, y sus observaciones y obtención de recompensas se deberían modificar. En el caso del agente perseguidor, se ha observado que el agente se suele mantener a más distancia de la que debería, perjudicando a las recompensas obtenidas. En el caso del agente que dispara, la mayoría de tiros fallan porque el agente no apunta correctamente al enemigo, quedándose unos grados a la izquierda. Resulta curioso si se compara con los resultados del agente de la galería de tiro, que tiene prácticamente los mismos parámetros de comportamiento, pero con un desempeño inferior.

Figura 3.4: *Recompensas acumuladas de los agentes del juego de caza*

3.1.2 Entrenamiento en escenario de Wave Function Collapse

Este ejemplo final se ha preparado de tal forma que cada entrenamiento se lleva a cabo en un mapa totalmente aleatorio al resto, pero basándose en el mismo modelo. De esta forma, nos aseguramos que los agentes no aprenden a adaptarse a un solo escenario completo. Se puede apreciar en la figura 3.5 que se han establecido 9 zonas de entrenamiento rectangulares, que toman de ejemplo el modelo en forma de cuadrado arriba a la izquierda.

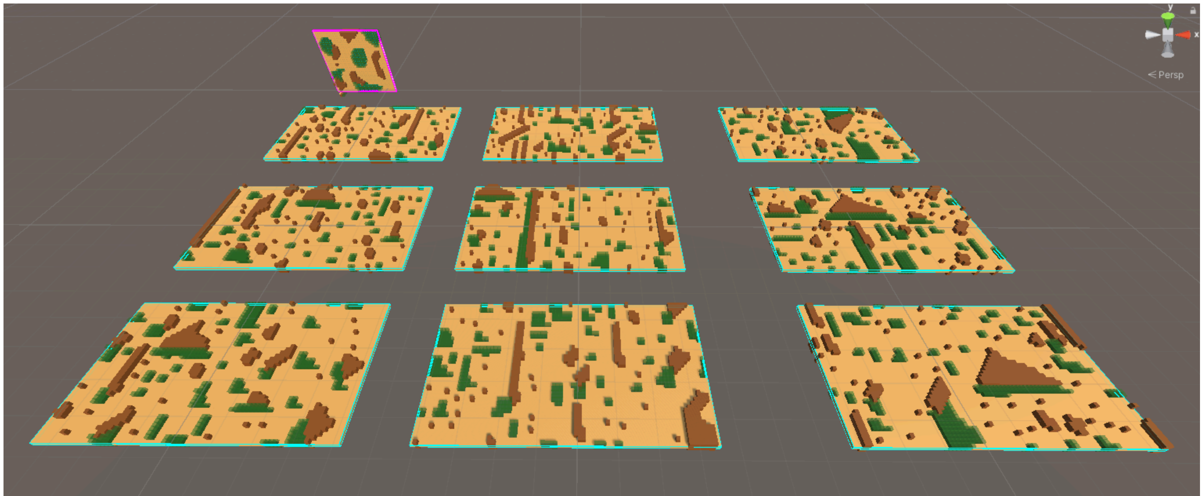


Figura 3.5: *Zonas de entrenamiento con Wave Function Collapse*

Dado que ahora los agentes se enfrentan a sus adversarios teniendo en cuenta los muros y montones de hierba, es esperable que su comportamiento se vea alterado respecto al ejemplo anterior, en el escenario simple. Igualmente, se ha modificado ligeramente las cantidades de recompensas obtenidas por el agente que patrulla para intentar mejorar su comportamiento. Concretamente, se ha reducido considerablemente sus recompensas obtenidas al estar cerca de su zona de aparición con el fin de reducirla, que era considerablemente mayor que la de sus iguales.

Esta vez, para poder apreciar los resultados, se mostrará la gráfica de aprendizaje de cada agente por separado, debido a las grandes diferencias existentes entre ellas, empezando por la del agente que patrulla en la figura 3.6.

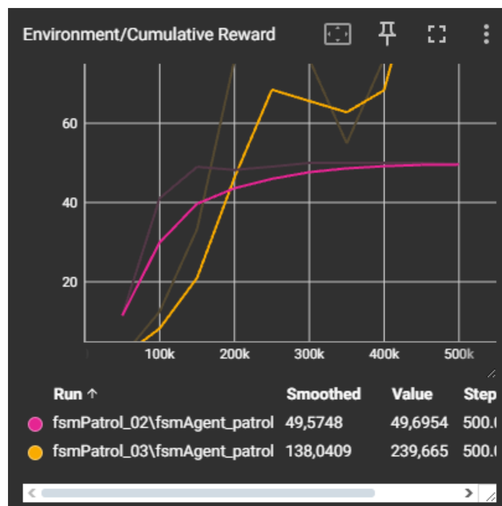


Figura 3.6: *Recompensas acumuladas del agente que patrulla*

A pesar de haber reducido las recompensas obtenidas, se puede observar cómo la recompensa acumulada es mucho mayor que en el escenario simple, tendiendo incluso a crecer mucho más. Como resultados, el agente se mantiene muy cerca de su posición de aparición, a diferencia de el escenario simple, donde solía dar vueltas más extensas y trataba de regresar si se alejaba demasiado. Esto puede ser debido a la existencia de los muros, que limitan su movimiento no solo por el contacto directo, sino por el hecho de intentar evitarlos.

Tanto el agente perseguidor como el que dispara no han recibido ninguna modificación en sus observaciones, pero sí se ha reducido ligeramente el ángulo respecto al enemigo con el objetivo de resolver el problema de apuntado del ejemplo del escenario sencillo. En este caso, se han visto reflejados ciertos cambios tanto en su aprendizaje como en su comportamiento final. La figura 3.7 muestra una comparativa entre entrenamiento entre el escenario simple y el final del agente que persigue, y la 3.8 de igual forma del agente que dispara.

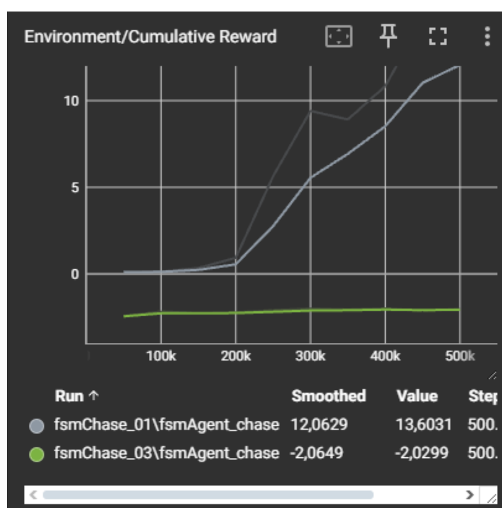


Figura 3.7: *Comparativa de recompensas acumuladas del agente que persigue*

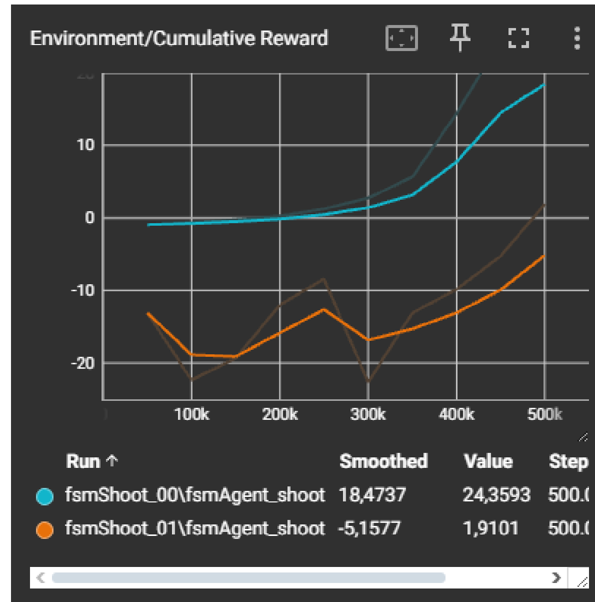


Figura 3.8: *Comparativa de recompensas acumuladas del agente que dispara*

Se puede ver en la figura 3.7 que ahora la ganancia de recompensas (representada por la curva verde) se estabiliza pero es prácticamente nula, manteniéndose siempre en negativo. Esto da a entender que el agente no es capaz de desempeñar mejor su tarea con su configuración actual, siendo los muros su principal impedimento. Por otro lado, el agente que dispara mantiene una curva de aprendizaje (naranja) con una tendencia de crecimiento similar a la del escenario simple (azul), pero con ciertas variaciones. De nuevo, los muros juegan un papel fundamental en su contra. Se ha podido apreciar que el agente ahora dispara con menor frecuencia, fallando menos tiros; pero a su vez, también deja pasar oportunidades que podrían ser un acierto sobre un enemigo. Es por ello, como se verá en el siguiente apartado y a pesar del intento de mejora en su apuntado, sigue fallando la mayoría de tiros. De este modo, se puede considerar que las recompensas acumuladas se deben principalmente a que el agente apunta correctamente a un enemigo, pero no porque consiga acertar.

3.1.3 Partidas simuladas

En este último apartado del desarrollo se van a llevar a cabo una serie de partidas simuladas, en primer lugar de manera independiente, y en segundo un agente contra el jugador. Se analizará la puntuación alcanzada en cada caso y la duración del episodio.

Este juego de caza se preparó con un agente con la función de cazador y siete enemigos repartidos por el mapa. Tras la ejecución de más de medio centenar de episodios, desgraciadamente los agentes no son capaces de alcanzar a ningún enemigo antes de que finalice el episodio en la mayoría de casos, que puede acabar ya sea porque el agente se salga de los límites del mapa, no haya más enemigos activos o que alcance

el número de pasos máximo, que es 7500. Este problema se debe principalmente a la navegación del agente que persigue al enemigo más cercano esquivando los obstáculos hasta estar a una distancia de tiro; y por otro lado al agente que dispara, que no logra posicionarse y apuntar correctamente para tener una trayectoria de tiro despejada. Se requeriría una revisión completa de la configuración de ambos agentes (observaciones, acciones y recompensas recibidas) para permitirles adaptarse mejor a estos escenarios aleatorios.

En cuanto al enfrentamiento contra el jugador, los resultados del agente cazador no difieren respecto a lo visto anteriormente. Su desempeño es muy limitado, siéndolo más aún cuando el jugador utiliza ciertas estrategias, como ignorar sus enemigos más cercanos y priorizar los de su rival para denegarle la obtención de puntuación. Cuando le es robado el enemigo al que perseguía, el cazador busca al siguiente más cercano. En este tiempo, el jugador puede aprovechar para repetir la misma jugada, y así hasta que finalice el juego. Por lo tanto, se puede determinar que el agente cazador no está actualmente a la altura de un jugador real en esta actividad competitiva, pues resulta sencillo sacar ventaja de sus limitaciones.

3.2 Tiempos de entrenamiento

Los entrenamientos se han ejecutado sobre un procesador AMD Ryzen 7 5800X 8-Core 3.80GHz, una GPU Nvidia RTX 3070 gaming z Trio 8GB y 32GB RAM, variando el número de escenarios de entrenamiento concurrentes dependiendo de la exigencia computacional, siendo un mínimo de nueve y un máximo de dieciséis.

A la hora del entrenamiento, ML-Agents permite duplicar el escenario para disminuir el tiempo del mismo. Esto se debe a que los agentes de estos escenarios están bajo una misma y única red neuronal. Por lo tanto, cuantos más escenarios de entrenamiento existan a la vez, más datos se van a volcar sobre la red en un mismo instante para modificar sus ponderaciones, disminuyendo el tiempo de entrenamiento. Sin embargo, esto a su vez puede aumentar el tiempo de ejecución si el número de escenarios de entrenamiento es demasiado alto, por lo que es importante buscar un equilibrio.

Cabe destacar la diferencia de tiempos de entrenamiento entre los escenarios simples y los generados con Wave Function Collapse, pues el procesamiento de todas las casillas generadas y las colisiones de los sensores de los agentes impactaban negativamente en el tiempo de ejecución. Esto resultó en la limitación del número de escenarios de entrenamiento, haciendo que el tiempo de entrenamiento aumentase considerablemente. Otro tiempo que llama la atención es de MoveToGoal con obstáculos, siendo el tiempo más elevado. Se debe a que se realizó una prueba para ver cuánto tardaría el entrenamiento con un solo escenario. La tabla 3.1 que se muestra a continuación muestra los tiempos de entrenamiento de cada agente.

Ejemplo	**Tiempo de entrenamiento (minutos)**
MoveToGoal	4.60
MoveToGoal (obstáculos)	32.16
Predator	23.58
Prey	25.85
Shooter	13.08
fsmPatrol (simple)	6.16
fsmPatrol (WFC)	21.32
fsmChase (simple)	6.86
fsmChase (WFC)	21.17
fsmShoot (simple)	7.44
fsmShoot (WFC)	25.41

Tabla 3.1: Tiempos de entrenamiento

Capítulo 4 Conclusiones y líneas futuras

Este proyecto ha servido como introducción tanto al kit de herramientas de ML-Agents como al algoritmo de Wave Function Collapse. El principal desafío encontrado ha sido el entrenamiento de los propios agentes, pues no siempre una modificación en las observaciones o las recompensas recibidas se traducen como se pudiera esperar en su comportamiento, resultando en numerosas ocasiones en modelos entrenados erráticos y que no se adaptan bien a los entornos generados aleatoriamente. Es un ámbito de estudio que requiere mucho tiempo, pero sin duda puede llegar a dar como resultado comportamientos profundamente complejos. Aparte, existen otros tipos de entrenamientos dentro del kit sobre los que no se ha llegado a trabajar.

En cuanto al algoritmo de Wave Function Collapse, su funcionamiento base es muy sencillo, pero los mapas generados pueden variar al tener un número escaso de tipos de casillas. En este proyecto no ha dado tiempo a desarrollar más casillas, pero sería una mejora notable tanto para el algoritmo como para los agentes inteligentes si estas dispusieran de alguna funcionalidad adicional, como la casilla “Grass”.

Como líneas futuras de investigación, la prioridad sería estudiar más a fondo el kit de herramientas de ML-Agents para resolver los problemas vistos de los agentes de este proyecto, y desarrollar agentes con comportamientos más complejos, capaces de hacer frente a jugadores haciendo uso de diversas estrategias.

Además, sería interesante estudiar la forma de implementar ambas partes en un sistema capaz de generar niveles de videojuegos con unas características determinadas, ya sea el tamaño del mapa, el número de enemigos o qué comportamiento tienen, pudiendo llamarse a estas características etiquetas. Estas etiquetas, a su vez, podrían ser clasificadas en función de una dificultad o desafío asociado. De esta forma, se facilitaría enormemente la generación potencialmente infinita de niveles aleatorios ya preparados para ser jugados por un usuario.

Capítulo 5 Summary and Conclusions

This project has served as an introduction to both the ML-Agents toolkit and the Wave Function Collapse algorithm. The main challenge encountered has been the training of the agents themselves, as not always a modification in the observations or rewards received translates as expected into their behaviour, resulting many times in erratic trained models that do not adapt well to randomly generated environments. It is a field of study that requires a lot of time, but undoubtedly can lead to deeply complex behaviours. Additionally, there are other types of training within the toolkit that have not been explored.

Regarding the Wave Function Collapse algorithm, its basic operation is straightforward, but the generated maps can vary when having a limited number of tile types. In this project, there was not enough time to develop more tiles, but it would be a significant improvement for both the algorithm and the intelligent agents if they had some additional functionality, such as the "Grass" tile.

As future lines of research, the priority would be to further study the ML-Agents toolkit to solve the problems seen in the agents of this project, and to develop agents with more complex behaviours, capable of facing players using diverse strategies.

It would also be interesting to study how to implement both parts in a system capable of generating video game levels with specific characteristics, whether it be the map size, the number of enemies, or their behaviour, which could be referred to as labels. These labels could, in turn, be classified based on an associated difficulty or challenge. This way, it would greatly facilitate the potentially infinite generation of randomly prepared levels ready to be played by a user.

Capítulo 6 Presupuesto

Para el presupuesto se toma de base un sueldo de 35€/hora, con una jornada de 8 horas, 5 días a la semana.

6.1 Apartados

Tipos	Horas	Coste
Documentación y preparación	80	2800
Desarrollo de casillas para WFC	5	175
Guardado y carga de mapas	15	525
Desarrollo y entrenamiento de los agentes	150	5250
Integración de la máquina de estados con los agentes	15	525
Integración y entrenamiento de los agentes con en los entornos generados con WFC	45	1575
Total	310	10850

Tabla 6.1: Resumen de tipos

La mayor parte del tiempo va destinado al entrenamiento de los agentes, pues cada modificación que se realizaba requería un plazo de unos minutos entrenando hasta ver si los resultados eran favorables o no. Esto da un total de 310 horas, que asciende el presupuesto hasta los 10850€.

Capítulo 7 Bibliografía

[1] *Unity Game Engine*. (2022). Unity-Technologies. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://unity.com/es

[2] *ML-Agents*. (2024). Unity-Technologies. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://github.com/Unity-Technologies/ml-agents

[3] M. Gumin. *Wave Function Collapse*. (2022). Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://github.com/mxgmn/WaveFunctionCollapse

[4] M. Gumin. *Unity - Wave Function Collapse*. (2019). Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://selfsame.itch.io/unitywfc

[5] O. Stålberg. *Townscaper* (2021). Raw Fury. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://store.steampowered.com/app/1291340/Townscaper/

[6] A. Stooke *et al.*, "Open-Ended Learning Leads to Generally Capable Agents", DeepMind, Londres, Reino Unido. Accedido: 11 de Julio, 2024. [En línea]. Disponible en: https://arxiv.org/pdf/2107.12808

[7] "An Introduction to Proximal Policy Optimization". machinelearningexpedition.com. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://www.machinelearningexpedition.com/ppo-proximal-policy-optimization/

[8] H. Kim, S. Lee, H. Lee, T. Hahn, S. Kang. (Agosto 2019). Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm. Presentado en 2019 IEEE Conference on Games. Disponible en: https://ieeexplore.ieee.org/abstract/document/8848019/authors#authors

[9] A. Sandhu, Z. Chen, J. McCoy. (Agosto 2019). Enhancing wave function collapse with design-level constraints. Presentado en FDG '19: Proceedings of the 14th International Conference on the Foundations of Digital Games. Disponible en: https://dl.acm.org/doi/abs/10.1145/3337722.3337752

[10] P. Thanh Hung, M. Duy Dan Truong, P. Duy Hung. (Julio 2022). Tuning Proximal Policy Optimization Algorithm in Maze Solving with ML-Agents. Presentado en International Conference on Advances in Computing and Data Sciences. Disponible en: https://link.springer.com/chapter/10.1007/978-3-031-12641-3_21

[11] "Proximal Policy Optimization". openai.com. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://openai.com/index/openai-baselines-ppo/

[12] Unity-Technologies. (2020). ML-Agents: Hummingbirds. [En línea]. Disponible en: https://learn.unity.com/course/ml-agents-hummingbirds

[13] *PyTorch*. (versión 2.3.1). PyTorch Foundation. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://pytorch.org

[14] "Wrappers - Gym Documentation". gymlibrary.dev. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://www.gymlibrary.dev/api/wrappers/

[15] R. Heaton. "The Wave Function Collapse Algorithm explained very clearly". robertheaton.com. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

[16] J. Parker. Unity Wave Function Collapse. (29 de Octubre, 2016). Accedido: 8 de Julio, 2024. [Vídeo en línea]. Disponible en: https://www.youtube.com/watch?v=CTJJrC3BAGM

[17] “What is reinforcement learning? | IBM”. ibm.com. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://www.ibm.com/topics/reinforcement-learning

[18] H. Francis Song *et al.*, “V-MPO: On-Policy Maximum a Posteriori Policy Optimization for Discrete and Continuous Control”, Universidad de Cornell, EEUU, 2019. Accedido: 11 de Julio, 2024. [En línea]. Disponible en: https://arxiv.org/abs/1909.12238

[19] “Unity ML-Agents Toolkit”. unity-technologies.github.io. Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://unity-technologies.github.io/ml-agents/

[20] Unity - Manual: ML Agents. (2022). Accedido: 8 de Julio, 2024. [En línea]. Disponible en: https://docs.unity3d.com/Manual/com.unity.ml-agents.html

Capítulo 8 Anexo

En este apartado se anexa un enlace al repositorio de GitHub que contiene el proyecto.