Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Deep Reinforcement Learning Aplicado al Testeo de Juegos

Deep Reinforcement Learning Applied to Game Testing

Alejandro Pérez Moreno

La Laguna, 08 de mayo de 2021

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **José Demetrio Piñeiro Vera**, con N.I.F. 43.774.048-B profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

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

Que la presente memoria titulada:

“Deep Reinforcement Learning Aplicado al Testeo de Juegos”

ha sido realizada bajo su dirección por D. **Alejandro Pérez Moreno**,

con N.I.F. 51.151.150-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 5 de mayo de 2021

Agradecimientos

A mis tutores, por ayudarme cuando lo he necesitado y por su paciencia.

Y a mi familia, por apoyarme siempre en los momentos de estrés.

Licencia

Resumen

El objetivo de este trabajo ha sido estudiar la viabilidad del uso de Inteligencia Artificial para comprobar el correcto funcionamiento de un videojuego.

Con la creciente complejidad en los videojuegos actuales y su desarrollo, realizar un testeo adecuado es cada vez un proceso más largo y costoso, lo que aumenta la necesidad de usar soluciones automatizadas. En la actualidad se usan principalmente probadores humanos y soluciones automatizadas basadas en script. Sin embargo, ambas soluciones presentan distintos problemas (determinismo, coste, tiempo…).

La solución propuesta consiste en implementar un modelo de Deep Reinforcement Learning (DRL) que aprenda a explorar el escenario, consiguiendo una mayor cobertura y eliminando el determinismo de otras opciones.

Palabras clave: Deep Reinforcement Learning, Testeo de juegos, Inteligencia Artificial

Abstract

The goal of this project has been to study the viability of the use of Artificial Intelligence for Game Testing.

With the increasing complexity in today's video games and their development, conducting proper testing is becoming a longer and more expensive process, which increases the need to use automated solutions. Currently, human testers and automated script-based solutions are mainly used. However, both solutions present different problems (determinism, cost, time ...).

The proposed solution consists of implementing a Deep Reinforcement Learning (DRL) model that learns to explore the scenario, achieving greater coverage and eliminating the determinism of other options.

Keywords: Deep Reinforcement Learning, Game Testing, Artificial Intelligence

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Introducción

En este trabajo se propone realizar un modelo general de Deep Reinforcement Learning (DRL) para el testeo de videojuegos, en particular aplicado a la navegación por el escenario.

Con la creciente complejidad de los videojuegos actuales, es necesario buscar alternativas automatizadas para su testeo. Para esto habitualmente se usan agentes basados en scripts, ya que son eficientes y fácilmente escalables. Sin embargo, dicho método presenta un alto grado de predictibilidad, y su capacidad de descubrir posibles fallos está limitada en parte por el conocimiento del juego de los desarrolladores, dificultando encontrar mecánicas no intencionadas. Usando DRL sin embargo, los agentes pueden aprender a explorar el escenario de formas más complejas e impredecibles, descubriendo potencialmente fallos fuera del alcance de otros métodos [1]. Combinando las ventajas del DRL con las de los agentes basados en scripts, se alcanzaría un alto grado de automatización a la vez que se obtienen pruebas de calidad.

El principal objetivo es desarrollar el modelo y probarlo en distintos escenarios con el fin de comprobar su efectividad. Adicionalmente se busca generar distintas visualizaciones que permitan sacar conclusiones a partir de los resultados obtenidos.

1.2 Objetivos

En este capítulo se exponen de forma más detallada cuáles son los objetivos de este trabajo.

1.2.1 Objetivo general

El objetivo general es diseñar un modelo genérico que sea capaz de explorar espacios tridimensionales y determinar la viabilidad del uso de DRL para el testeo de videojuegos a través de dicho modelo. También se pretende explorar otras opciones más específicas a fin de potenciar ciertos tipos de exploración.

1.2.2 Objetivos específicos

- Desarrollar una serie de escenarios a testear.
- Decidir los sensores necesarios para un modelo genérico.

- Diseñar una función de recompensa apropiada para el modelo.
- Obtener diagramas y visualizaciones útiles para analizar los resultados de la exploración del escenario.
- Probar distintas configuraciones para agentes más específicos (módulos de curiosidad, agente visual…).

Capítulo 2

Estado del Arte

En el capítulo anterior se ha realizado una breve introducción de las ideas y objetivos principales de este trabajo, así como un repaso por algunos de los antecedentes en este tema.

En este capítulo se profundizará en las diferentes tecnologías usadas durante el desarrollo, así como trabajos previos relacionados.

2.1 Deep Reinforcement Learning

El Deep Reinforcement Learning es un campo dentro de la inteligencia artificial, y más específicamente dentro del Machine Learning. Como su nombre indica, es una fusión de dos técnicas distintas: El Deep Learning y el Reinforcement Learning (RL) [2].

2.1.1 Deep Learning

Aunque el Deep Learning no es una parte central de este trabajo, es importante entender algunas de sus características. El Deep Learning es un conjunto de algoritmos de Machine Learning caracterizados principalmente por una estructura computacional formada por múltiples capas. Esto se traduce en un procesamiento más profundo de las entradas para producir las salidas, lo que permite aprender patrones y estructuras más complejas, y con mayor cantidad de entradas. Las estructuras usadas en estos algoritmos son conocidas como Deep Neural Networks, y un ejemplo de estas son las Convolutional Neural Networks, usadas principalmente en el procesamiento de imágenes.

2.1.2 Reinforcement Learning

Para entender el Deep RL es fundamental entender en primer lugar el RL. DRL es un tipo de RL en el que, para representar la política actual, se utilizan Deep Neural Networks.

Un modelo de RL está compuesto por varios elementos:

- El **entorno**, donde el agente realiza sus acciones.
- El **agente** es la entidad que realiza las acciones.
- El conjunto de posibles **estados** en los que el agente se puede encontrar. Estos estados están caracterizados por la forma en que el agente percibe el entorno.
- El conjunto de **acciones**, que el agente puede tomar en función del estado.
- La **política** (*policy* en inglés) mapea un estado a una posible acción a tomar.
- El **algoritmo** determina cómo evoluciona la política y marca en última instancia el aprendizaje. A los parámetros que determinan el comportamiento específico del algoritmo se los conoce como **hiperparámetros.**

Al usar RL se entrena un modelo para tomar una serie de decisiones (o acciones) en función de una o más entradas. Aunque existen métodos que usan modelos para

aprender, lo más habitual es que la forma de aprendizaje sea mediante prueba y error. La forma de controlar su comportamiento es mediante señales de recompensa, que premian o castigan ciertas conductas del agente. Su objetivo es maximizar esta recompensa.

En cuanto al algoritmo, en este trabajo sólo se hará referencia al algoritmo Proximal Policy Optimization (PPO) [3]. Se ha decidido usar PPO por su sencillez y robustez frente a cambios en los hiperparámetros, lo que facilita el perfeccionamiento de los modelos.

2.2 Entorno de Trabajo

Para el desarrollo de este proyecto se han utilizado principalmente dos herramientas: el motor Unity, y ML-Agents.

2.2.1 Unity

Unity se trata de uno de los motores de videojuegos más usados en la actualidad [4]. Cuenta con un editor propio en tiempo real y gran cantidad de recursos. Además, dentro del propio Unity Editor se encuentra Collab, que se ha usado para mantener el proyecto en la nube.

2.2.2 ML-Agents

ML-Agents es un conjunto de herramientas para el entrenamiento de agentes de RL en Unity y está formado por dos componentes principales: un paquete de Unity y un módulo de Python.

La forma más sencilla de interactuar con ML-Agents es mediante *mlagents-learn*, una utilidad que permite hacer uso de los algoritmos disponibles para entrenar agentes de Unity. Es desde este punto de acceso desde donde se pueden especificar los hiperparámetros para el algoritmo mediante un fichero de configuración *.yaml*, así como en qué dispositivo se desea ejecutar (CPU o GPU). Al finalizar el entrenamiento almacena la red neuronal entrenada en un fichero ONNX [5].

2.3 Trabajos Previos

En la línea de automatizar el testeo de videojuegos, una de las soluciones más comunes es usar modelos diseñados a mano, haciendo uso de scripts. Esta opción, aunque sencilla y escalable, carece de la adaptabilidad necesaria para descubrir ciertos fallos en las mecánicas del juego. Respecto al uso de Deep Learning, este se ha usado tanto para el testeo 2D [9] como en algunos juegos de combate.

Por último, la principal inspiración para realizar este trabajo viene del paper “Augmenting Automated Game Testing with Deep Reinforcement Learning” [12], en el que se estudia la creación de un framework para testeo de videojuegos usando DRL enfocado a la exploración.

Capítulo 3 Desarrollo

En este capítulo se describe el proceso de desarrollo seguido hasta obtener el resultado final.

3.1 Escenarios

Se han creado 4 escenarios con diferentes objetivos. Para ello se han utilizado las herramientas ProBuilder y ProGrids dentro de Unity. Todos los escenarios diseñados constan de tres objetivos que el agente debe alcanzar en un determinado orden.

3.1.1 Escenarios de prueba iniciales

Los primeros escenarios desarrollados fueron escenarios de prueba, con el objetivo de familiarizarse con Unity, ProBuilder y ML-Agents. Este primer escenario tiene 4 objetivos, y los agentes pueden completarlo simplemente yendo en línea recta hacia el siguiente, por lo que es muy sencillo.

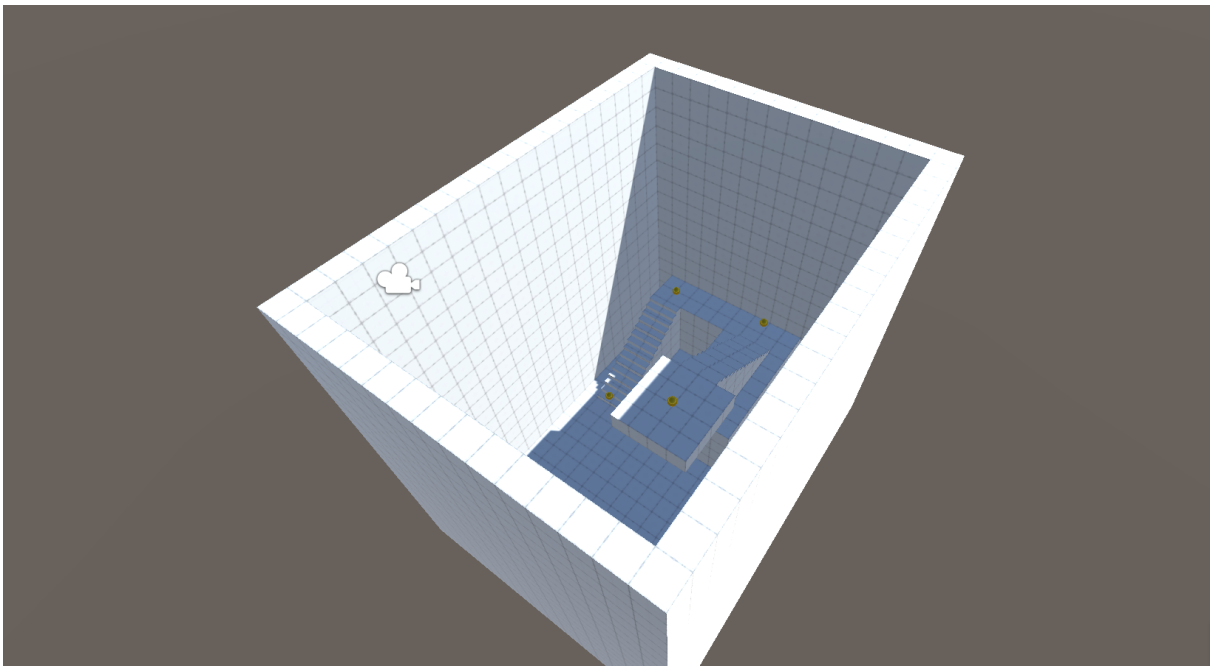

***Figura 1**. Escenario de pruebas 1*

Para comprobar la viabilidad de que los agentes aprendan a saltar en las primeras fases del desarrollo, se diseñó un escenario en el que los agentes tienen como único objetivo saltar de una plataforma a otra. Para simplificar su tarea, ambas están a una distancia corta y hay paredes para orientar la dirección correcta a los agentes. También fue en este punto cuando se incorporó la idea de que los agentes pudieran caer el vacío y ser penalizados por ello.

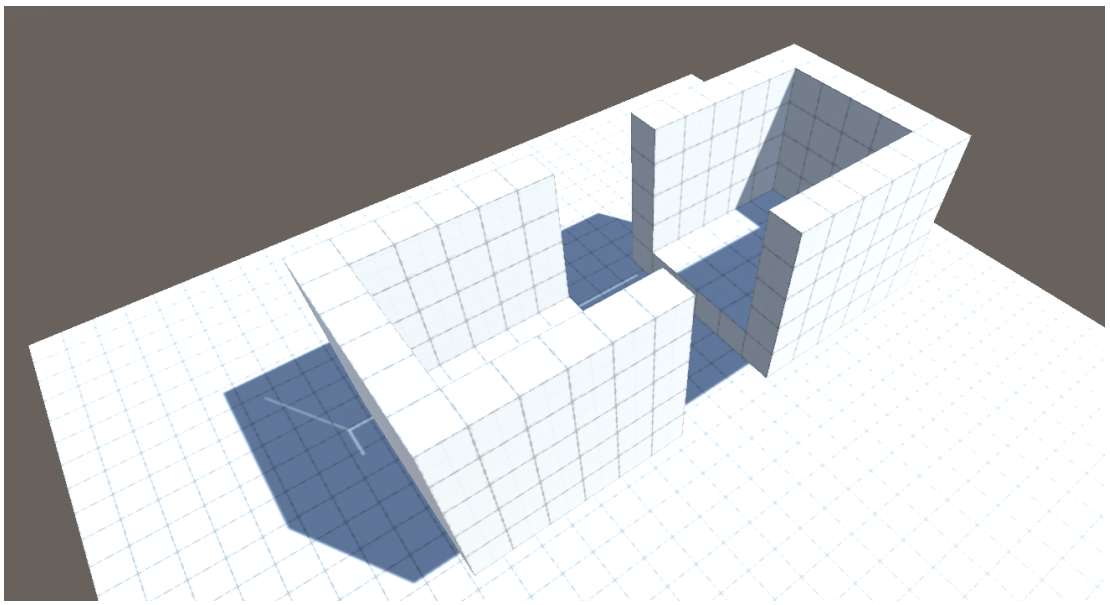

Figura 2. *Escenario de prueba salto*

3.1.2 Exploración con muro sin colisión

El primer escenario real consiste en un entorno en el que los agentes deben subir una serie de escalones para conseguir las recompensas. En este caso, se ha eliminado la malla de colisiones de uno de los muros intencionadamente con el objetivo de comprobar si los agentes son capaces de encontrar este fallo.

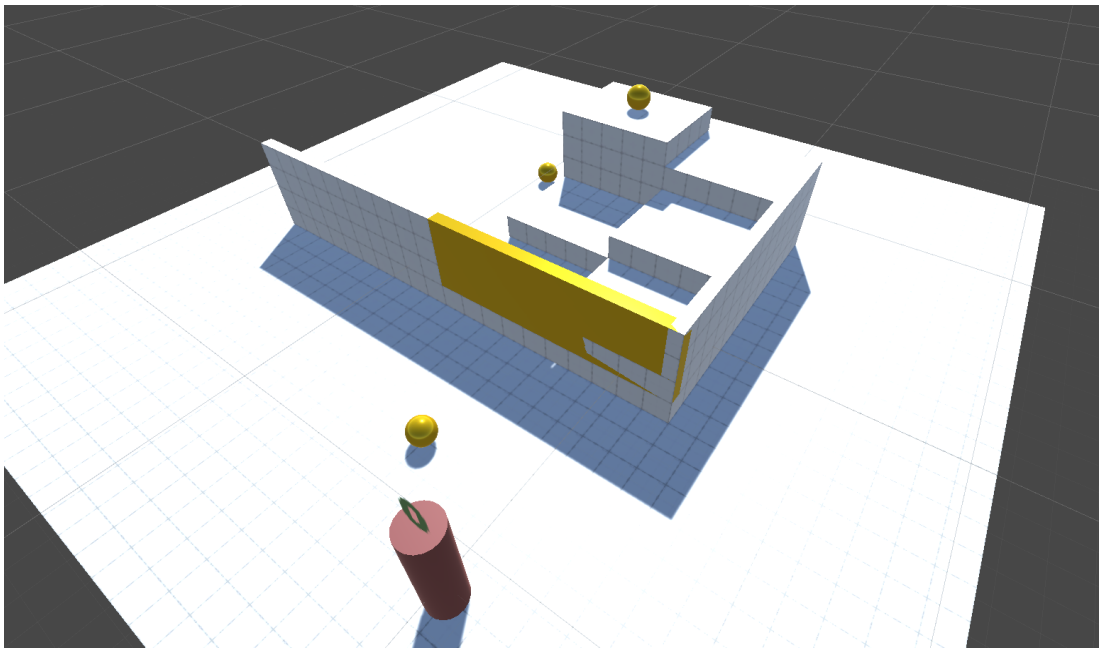

Figura 3. *Escenario con muro sin colisión (en amarillo)*

3.1.3 Exploración con zonas de bloqueo

En este escenario, en lugar de evaluar el comportamiento de los agentes se pretende evaluar la capacidad de generar reportes útiles que ayuden a interpretar los resultados de la exploración.

El escenario es el mismo que el anterior, pero se han añadido zonas en las que el agente puede quedar bloqueado si entra, para representar posibles fallos de este tipo en el juego.

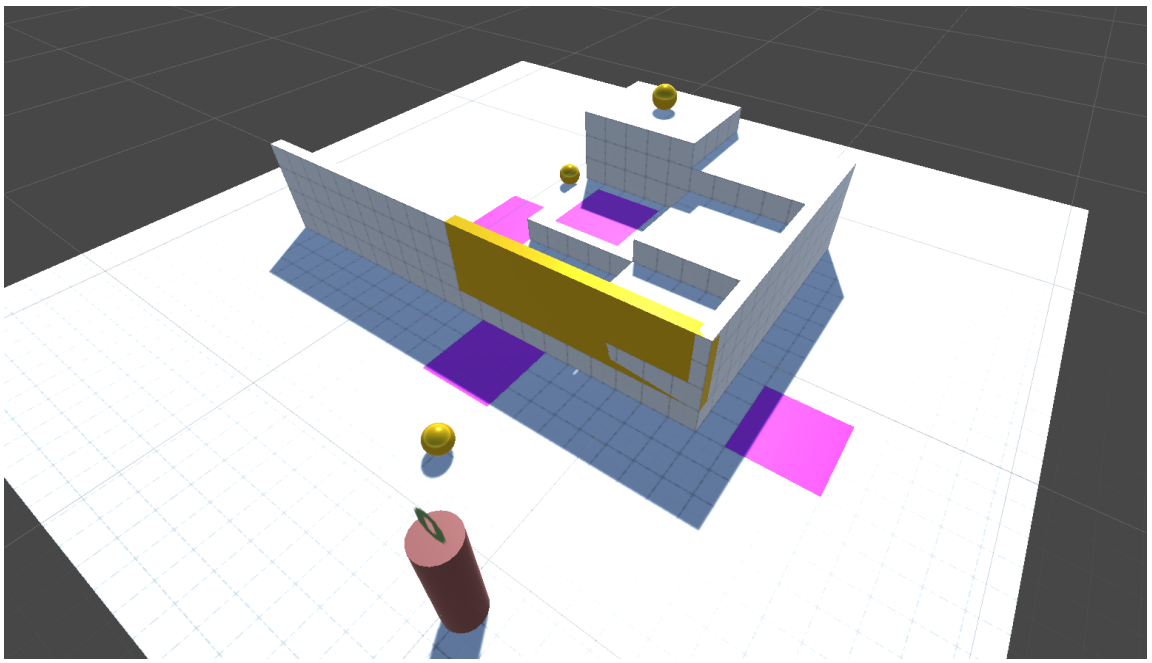

Figura 4. *Escenario con zonas de bloqueo (en violeta)*

3.1.4 Plataformas y movimiento

Para evaluar la capacidad de resolver problemas más complejos, se ha diseñado este escenario en el que los agentes deben recorrer una serie de plataformas saltando entre estas. La dificultad es incremental, ya que los saltos son cada vez más altos y las plataformas más pequeñas. Por último, se ha incluido una plataforma móvil como una primera aproximación a escenarios dinámicos.

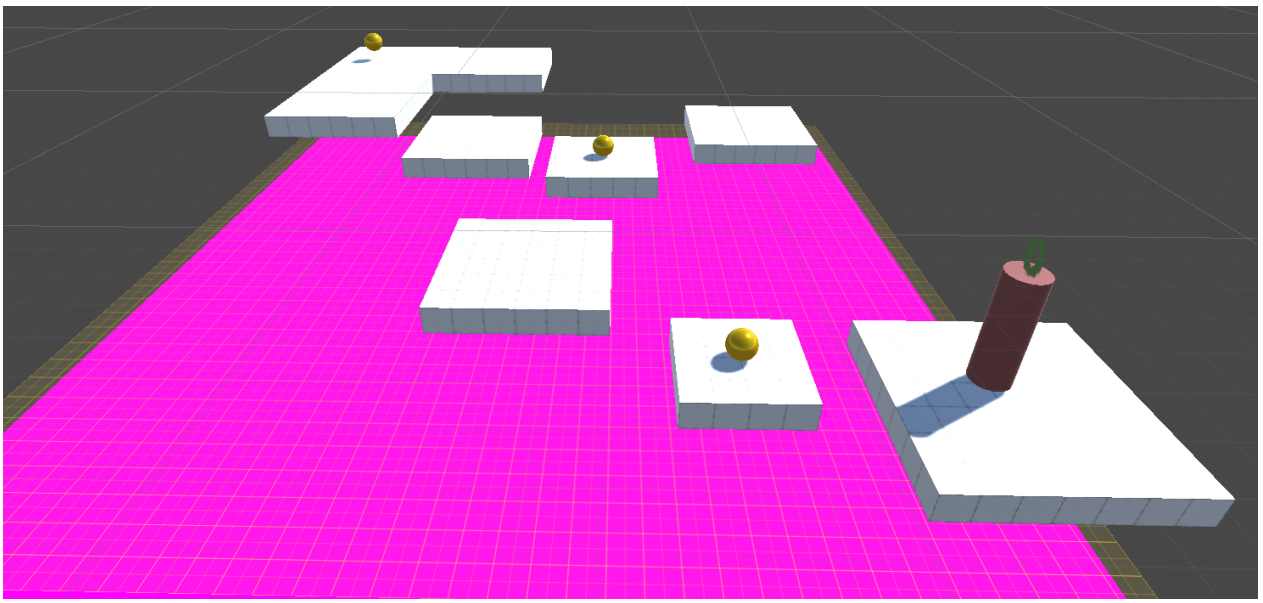

Figura 5. *Escenario de salto y plataforma móvil*

3.1.5 Precisión y espera

Este último escenario se desarrolló al observar que en los anteriores los agentes siempre podían alcanzar su objetivo yendo a la velocidad máxima y sin detenerse.

Por tanto, en este se introdujo una plataforma móvil por la que los agentes deberán esperar a veces, así como una pasarela estrecha para comprobar la capacidad de los agentes de ejecutar movimientos con precisión.

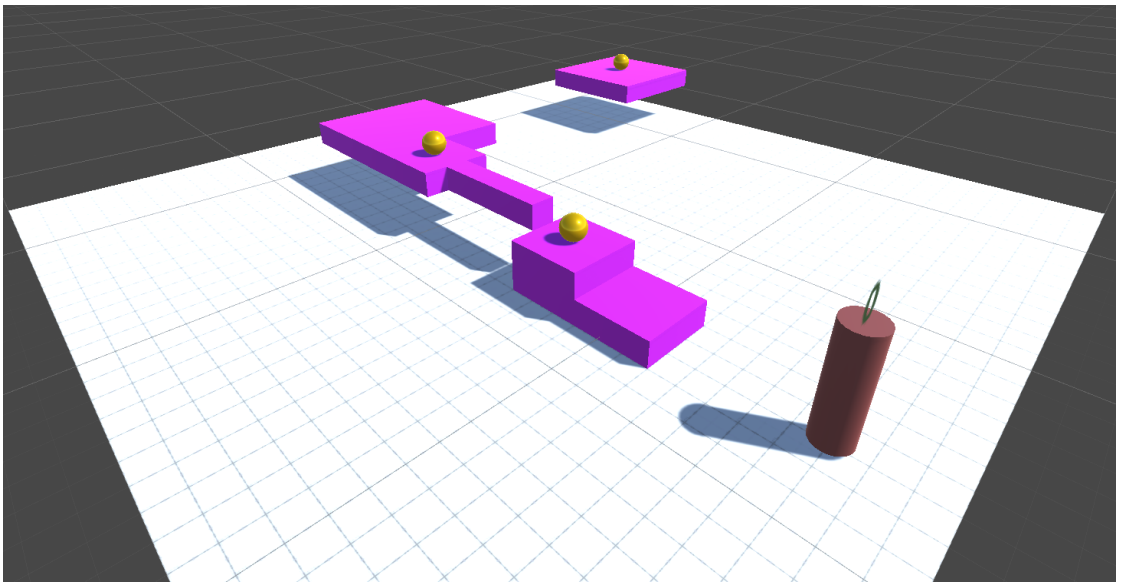

Figura 6. *Escenario de precisión y espera*

3.2 La función de recompensa

La función de recompensa es un componente que marca en gran medida la velocidad de entrenamiento y que, al modificarla, permite decidir de forma indirecta cómo se comportan los agentes.

Para este caso la señal de recompensa más básica en la que se puede pensar consiste en dar una recompensa positiva cada vez que un agente alcanza un objetivo. De esta forma, los agentes aprenden a ir tras ellos. Sin embargo, de usar dicha función nos encontraríamos con un problema conocido dentro del RL: el problema de *Sparse Rewards* o recompensas dispersas [6], que implica que si sólo se recompensa a los agentes por alcanzar un objetivo, la mayor parte del tiempo no obtendrán recompensa y por tanto el algoritmo no tendrá forma de decidir si las acciones tomadas son buenas o no. Esto provoca que los periodos de entrenamiento sean más largos y dependientes de la aleatoriedad.

En este trabajo los agentes serán entrenados por cada escenario hasta que sean capaces de resolverlo, por lo que no es necesario que un agente ya entrenado se adapte a situaciones nuevas. Teniendo esto en cuenta es posible usar la solución más sencilla al problema de Recompensas Dispersas: *Reward Shaping*. La premisa del Reward Shaping es que, partiendo de la concepción como humanos del problema, es posible diseñar una función que recompense al actor de forma más frecuente (a menudo tras cada acción tomada), premiando no sólo puntos clave como alcanzar un objetivo sino también pequeñas acciones que acerquen al agente a alcanzarlos.

En este caso se ha optado por recompensar al agente por cada paso en la dirección del siguiente objetivo, y dar una recompensa negativa de la misma magnitud cuando el agente dé un paso que lo aleje de la siguiente meta. Además, en algunos escenarios los agentes pueden caer al vacío. Los agentes que caigan también serán penalizados. Así, la función de recompensa final tiene la siguiente forma:

- Por cada paso que acerque/aleje el agente al siguiente objetivo: +0.01/-0.01.
- Por cada objetivo alcanzado: +1.
- Por cada ocasión en que el agente caiga: -1.

Estos valores se han ido ajustando mediante prueba y error, observando los cambios en el comportamiento de los agentes al modificar la señal de recompensa. De este proceso se han extraído una serie de conclusiones relevantes a la hora de diseñar esta función.

La primera conclusión es que los agentes no responden correctamente a señales de recompensa elevadas. Originalmente estas señales de recompensa eran aproximadamente 3 órdenes de magnitud mayores. Esto hacía que el algoritmo no funcionase de forma eficiente, por lo que se optó por reducir la magnitud de las recompensas.

Además, se concluyó que penalizar en exceso ciertas acciones puede llevar a situaciones contraproducentes en las que el agente queda paralizado en un *mínimo local*, ya que

desarrollaba miedo a explorar por ser penalizado. Esto se observó en las primeras iteraciones de la función de recompensa, en las que alejarse se penalizaba el doble de lo que se recompensaba al acercarse, y de forma similar, al penalizar en exceso el caer al vacío. Esto restringía en gran medida el interés de exploración del agente.

3.3 Agente

Otra parte crucial de este trabajo es el desarrollo de un agente que sea capaz de resolver los problemas sin que el desarrollador invierta demasiado esfuerzo en el proceso de entrenamiento.

Diseñar un agente equivale a decidir los siguientes puntos:

- **Geometría del agente.** En este caso, la geometría es siempre la misma que la de un posible jugador, usando una malla de colisión cilíndrica.
- **Acciones**. Representan qué cosas puede hacer el agente. Dado el objetivo de los entrenamientos, es importante que estas acciones estén mapeadas de forma directa a las entradas que podría proporcionar un usuario. Esto es, dos ejes para el movimiento hacia delante y los lados y una acción de salto. El control de cámara ha sido eliminado en la mayoría de agentes por simplificar su control.
- **Sensores.** Es este punto el que más modificaciones sufre a lo largo del desarrollo. Los sensores representan la forma en que el agente percibe el escenario.

A continuación se detallan las diferentes fases en el proceso de desarrollo del agente hasta alcanzar el resultado final.

3.3.1 Agente básico

Este primer agente es el más sencillo tanto en cuanto a acciones como a sensores. Los ejes de movimiento son representados por valores discretos (-1, 0 o 1) de forma similar a cómo se haría con un control por teclado. El salto es una acción binaria, que representa si el agente debe saltar (1) o no (0).

En cuanto a los sensores, este agente sólo conoce su posición y la de las tres recompensas.

3.3.2 Agente con movimiento continuo

El siguiente paso fue cambiar las acciones de movimiento por valores continuos en lugar de discretos. Con esto, se busca alcanzar la libertad de movimiento que ofrece un joystick, pudiendo controlar no sólo la dirección sino también la velocidad.

3.3.3 Agente con sensores adicionales

Aunque el agente anterior era capaz de resolver escenarios sencillos como el escenario de prueba, el reducido número de sensores limitaba su efectividad. Por ello, los sensores anteriores fueron cambiados y ampliados por los siguientes:

- Posición Relativa del Agente respecto a la siguiente recompensa.
- Distancia entre el agente y la siguiente recompensa. Es importante incluir un sensor que se mapee de forma directa a una métrica de la señal de recompensa, como es la distancia.
- Si el agente se encuentra saltando o no.
- Tiempo restante en el episodio.

3.3.4 Agente con entradas normalizadas

Al probar el agente anterior, aunque eventualmente aprendía a resolver los escenarios, al inicio tenía una cierta tendencia a moverse hacia un lado en específico. En la búsqueda de a qué se podía deber se encontró un potencial problema: las entradas no estaban normalizadas [7]. Normalizar las entradas en RL implica limitar el rango de valores que estás pueden tomar, idealmente haciendo que la media de los valores de cada entrada sea 0. Para conseguir esto se puede hacer uso de un parámetro en ML-Agents. Sin embargo, esto ralentiza los primeros episodios del entrenamiento ya que el algoritmo debe determinar cuáles son los valores máximos, mínimos y medios de cada entrada. Por tanto se ha optado por normalizar los valores a mano entre -1 y 1. Con esto, se consigue reducir los tiempos de entrenamiento y mejorar la convergencia de estos.

3.3.5 Agente con RayCast

Hasta este punto, el agente no tenía ninguna forma de sentir realmente su entorno. Los sensores eran más bien conocimientos que este tenía. Esto provocaba que, incluso estando completamente entrenado, si el agente se desviaba ligeramente de su ruta a menudo permanecía colisionando con un muro hasta que el tiempo del episodio se agotaba. Para mitigar este problema se introdujeron un conjunto de 12 RayCasts. Estos son sensores que se proyectan desde el cuerpo del agente y permiten detectar objetos próximos a él.

3.3.6 Agente con RayCast mejorado

Aunque la versión anterior del agente podía resolver el escenario inicial de Exploración, era incapaz de avanzar en el escenario de plataformas. Para intentar corregir este problema se cambió la distribución de los RayCasts de forma que 6 de los 12 estuvieran dirigidos hacia el suelo. De esta forma, el agente puede determinar cuándo la plataforma en que se encuentra termina, si hay alguna plataforma próxima o la presencia de un escalón. En el siguiente diagrama se puede observar dicha distribución y como permite al agente detectar situaciones muy variadas.

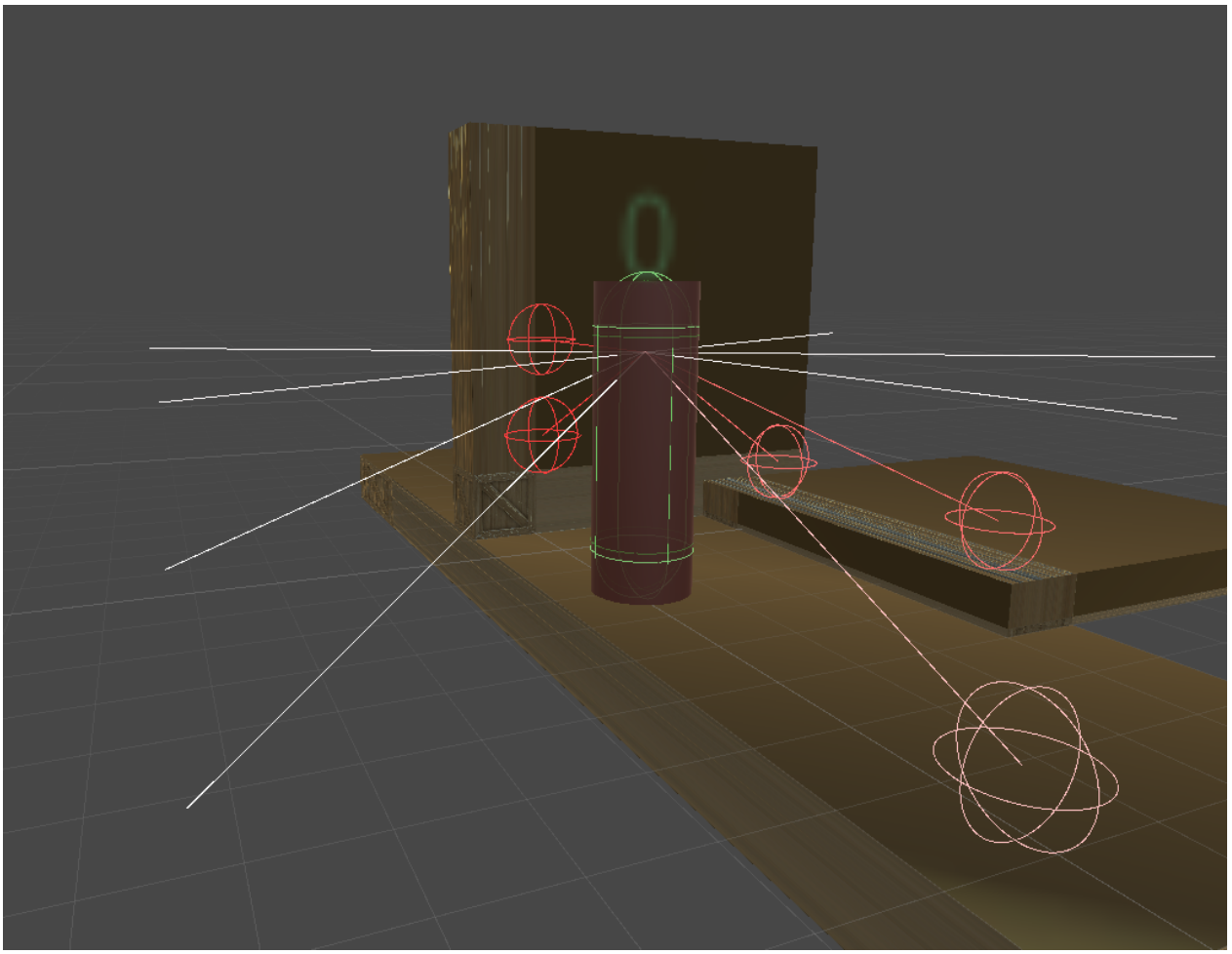

Figura 7. *Muestra de sensores del agente en entorno de prueba*

En este caso los RayCasts son representados con las líneas que se proyectan desde el cuerpo del agente (cilindro marrón). Nótese que los RayCast no actúan como sensores binarios, sino que proporcionan un valor en función de la proximidad de la colisión con un objeto. Esto se representa en la imagen mediante las distintas tonalidades de rojo, representando un color más intenso una colisión próxima y el blanco la ausencia de colisión. Se puede observar que gracias al ángulo en que se proyectan los RayCasts el agente puede determinar si la plataforma termina o no, si hay un escalón, o si se encuentra frente a un muro.

3.3.7 Otros agentes

Adicionalmente, se han realizado pruebas con agentes distintos, como agentes visuales, o que hacen uso del *módulo de curiosidad*. En el apartado 3.5 se discuten más en detalle estos agentes.

3.4 Configuración y Metodología de las pruebas

Dado que el objetivo del desarrollo es construir un modelo que pueda adaptarse a distintos escenarios con mínimo o nulo esfuerzo por parte del desarrollador, la configuración se ha mantenido de forma similar durante la mayor parte de las pruebas. El algoritmo usado es PPO, precisamente con el objetivo de evitar invertir excesivo tiempo haciendo *Hyperparameter Tuning*. Para definir los hiperparámetros se debe usar un fichero *yaml*. A continuación se encuentra el fichero usado:

```
behaviors:
    BehaviourName:
        trainer_type: ppo
        hyperparameters:
            batch_size: 1024
            buffer_size: 10240
            learning_rate: 3.0e-4
            beta: 0.005
            epsilon: 0.2
            lambd: 0.95
            num_epoch: 3
            learning_rate_schedule: linear
        network_settings:
            normalize: false
            hidden_units: 128
            num_layers: 2
        reward_signals:
```

```
        extrinsic:
            gamma: 0.99
            strength: 1.0
    max_steps: 20000000
    time_horizon: 64
    summary_freq: 50000
```

En ML-Agents, los agentes operan en pasos. En cada paso el agente recoge las observaciones del entorno, las pasa a la política, y recibe una acción en respuesta. Para realizar las pruebas, el entrenamiento se ha realizado en 20 millones de pasos. Cada episodio tiene una duración de 2000 pasos como máximo, para evitar que los agentes queden atrapados en mínimos locales.

Una vez realizado el entrenamiento inicial se han terminado de perfeccionar los modelos con un segundo entrenamiento de 20 millones de pasos, inicializado a partir del anterior.

3.5 Pruebas adicionales

En este apartado se comentan de forma más extensa los agentes y pruebas adicionales realizadas.

3.5.1 Agente Visual

En la búsqueda de la combinación ideal de sensores y partiendo de la premisa de que el agente debe actuar de la forma más similar posible al futuro jugador, se decidió probar a sustituir los sensores mencionados anteriormente por un único sensor en forma de cámara. De esta forma la percepción del escenario por parte del agente ocurre de forma visual al igual que ocurriría con un usuario del videojuego.

La entrada proporcionada al agente es una imagen de 84 por 84 píxeles, capturada desde una posición cercana a donde se encontraría la cámara del jugador. La resolución escogida permite al agente interpretar su entorno de manera suficiente para aprender, pero sin incurrir en el coste computacional y de almacenamiento que supondría usar una imagen de alta definición como la que vería el jugador. Esta entrada es procesada por un encoder compuesto por 3 capas convolucionales [8].

La configuración de este agente es similar al original, pero se debe añadir el tipo de *encoder* a usar. Basta con añadir este bloque al fichero *yaml* original:

```
behaviors:
    (…)
    network_settings:
        (...)
        vis_encode_type: nature_cnn
    (...)
```

Como se comentó anteriormente, el *encoder* utilizado es el que ML Agents llama *nature_cnn* llamado así por estar basado en un trabajo publicado en *Nature* [8], que consta de 3 capas convolucionales.

3.5.2 Módulo de curiosidad

Como se comentó en el punto 3.2, en este caso para solucionar el problema de Recompensas Dispersas se ha usado Reward Shaping. Aunque es cierto que en este caso la adaptabilidad del modelo ya entrenado no es un factor relevante, tal y como se ha diseñado, la función de recompensa incita a ir de forma directa al siguiente objetivo, como si se trazase una línea recta hacia él que el agente debe aprender a seguir.

Sin embargo hay otra solución a este problema, que además se asemeja más a la forma en que un humano explora: El uso de Curiosidad [9]. Esta técnica busca solucionar el problema de recompensas dispersas dando periódicamente recompensas que premian la curiosidad o exploración. Más concretamente, en ciertos momentos se premia que el agente tome acciones inesperadas, distintas a las predichas con mayor probabilidad por el modelo. Con esto se consigue que el agente pruebe nuevas acciones.

En ML Agents se incluye una opción llamada Módulo de Curiosidad para activar esta señal de recompensa. Para activarla simplemente hay que añadir el siguiente bloque al fichero *yaml* original:

```
behaviors:
    (...)
    reward_signals:
      extrinsic:
        (...)
      curiosity:
        strength: 0.8
        gamma: 0.99
```

Nótese que este bloque se añade dentro de las señales de recompensa, pero fuera de la sección "*extrinsic*". Esto es porque existen dos tipos de señales de recompensas: extrínsecas, que son proporcionadas por la interacción con el entorno (por ejemplo, alcanzar un objetivo o acercarse a él) e intrínsecas, que son proporcionadas por el propio modelo para ayudar a alcanzar el objetivo. La señal de curiosidad es por tanto una señal intrínseca. Un parámetro importante a tener en cuenta es el de *strength*. Este parámetro se debe ajustar a través de varias pruebas, y en esencia condiciona la intensidad de la señal de curiosidad. Esto es fundamental para evitar que la señal de curiosidad tenga efectos nulos o que, por el contrario, acabe abrumando a las señales extrínsecas. En este caso se ha decidido usar 0.8, ligeramente inferior al valor por defecto 0.9.

Capítulo 4 Resultados

En este capítulo se presentan los resultados obtenidos en los distintos entrenamientos.

4.1 Resultados Estadísticos

A continuación se presentan los tiempos de entrenamiento y datos estadísticos de cada escenario. Para interpretar los resultados es necesario tener en cuenta varios puntos. En todas las gráficas el eje de abscisas representa el número de pasos dados por los agentes. En algunas sesiones se interrumpió prematuramente el entrenamiento debido a que los agentes ya habían aprendido a resolver el escenario correctamente antes de los 20 millones de pasos.

Las gráficas con nombre “Cumulative Reward” representan en el eje de ordenadas la recompensa acumulada media de los agentes. La gráfica “Episode Length” representa el tiempo que tardan los agentes en completar cada episodio, medido en número de pasos. Un episodio puede terminar porque se agote el tiempo establecido (2000 pasos) o porque el agente caiga al vacío.

4.1.1 Escenarios de Exploración

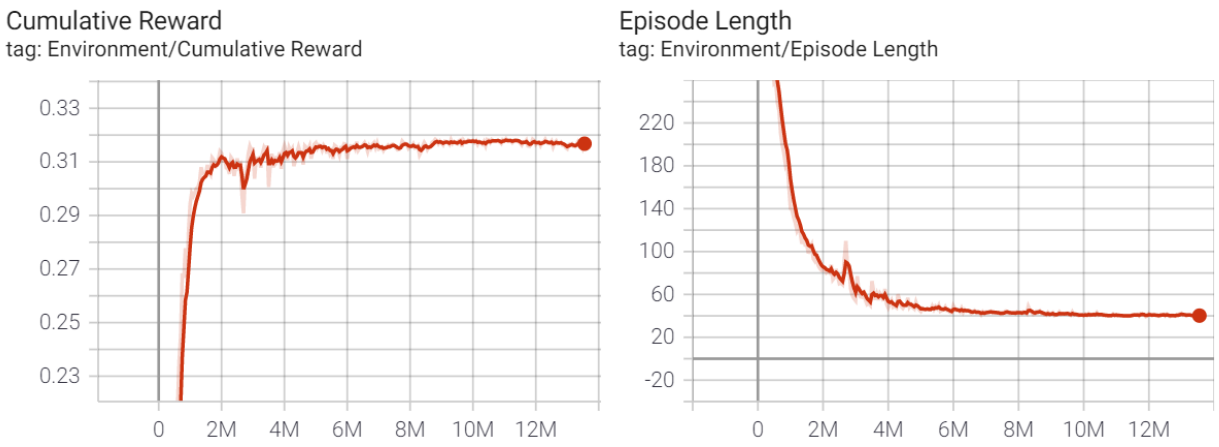

Figura 8. *Gráficas para el escenario de exploración*

Como se puede observar en las gráficas, para este primer escenario la recompensa conseguida por los agentes crece de forma rápida. En este caso se han usado 9 agentes simultáneos.

Tras dos millones de pasos (unos 30 minutos) se alcanza un punto muy cercano a la recompensa máxima conseguida. En los pasos restantes se mejora principalmente la estabilidad. Por otro lado, en torno a los seis millones de pasos (1 hora y 20 minutos) los agentes alcanzan una velocidad de resolución próxima a la máxima. El tiempo total, tras 13.55 millones de pasos ha sido de 3 horas y 3 minutos.

En este primer entrenamiento los agentes ya conseguían resolver el escenario de forma

consistente. Para terminar de perfilar el modelo, se realizó una segunda sesión de entrenamiento de 11.25 millones de pasos (2 horas y 25 minutos). En esta segunda iteración se buscaba principalmente que los agentes fueran más versátiles, ya que en algunas ocasiones con el primer entrenamiento si un agente se desviaba mínimamente de una ruta óptima, no era capaz de recuperarse y quedaba atascado hasta que se agotase el tiempo.

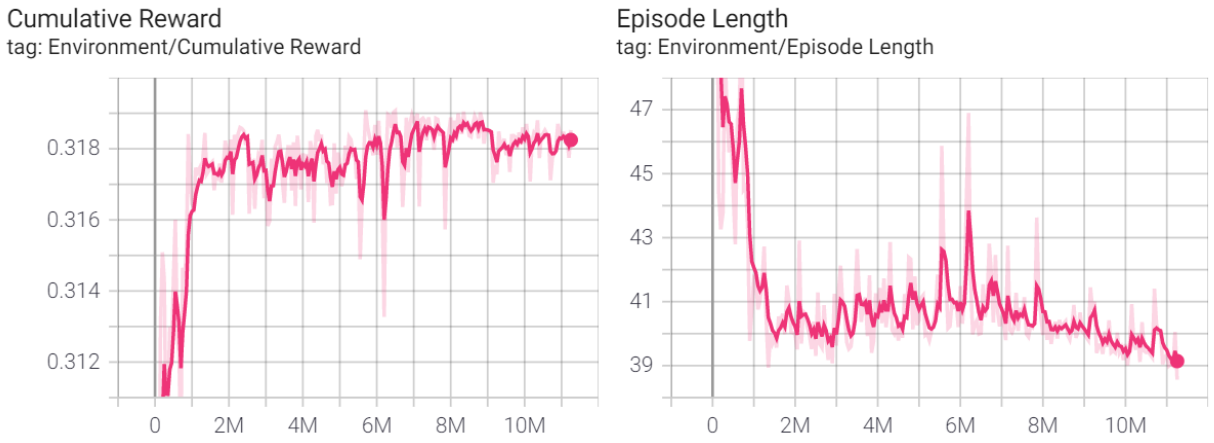

Figura 9. *Gráficas del segundo entrenamiento en el escenario de exploración*

Estos objetivos se han alcanzado de manera satisfactoria. También se ha mejorado ligeramente la recompensa y el tiempo. Nótese que la escala es distinta a la de las primeras gráficas. Estas son las dos gráficas de recompensa superpuestas:

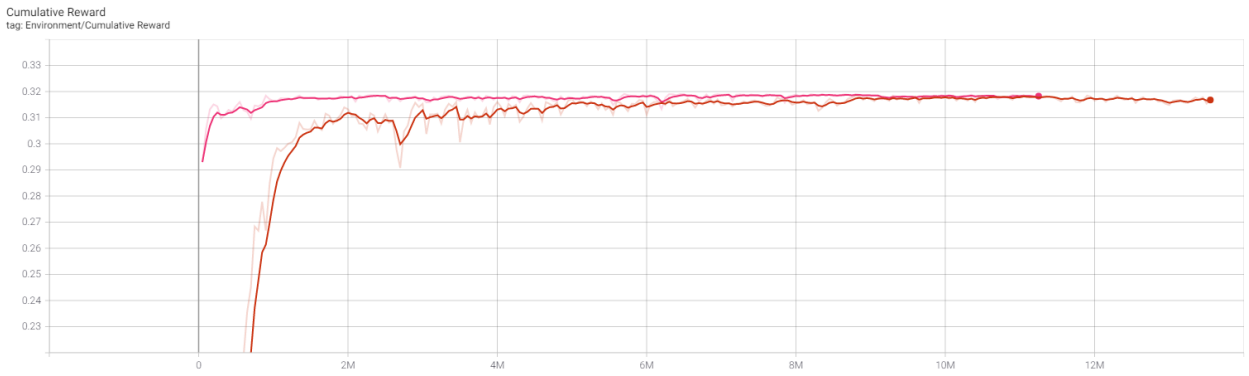

Figura 10. *Gráficas superpuestas*

4.1.2 Escenario Plataformas y Movimiento

La mayor complejidad del segundo escenario se ha reflejado en los tiempos de entrenamiento. En este caso se han utilizado 20 agentes de forma simultánea.

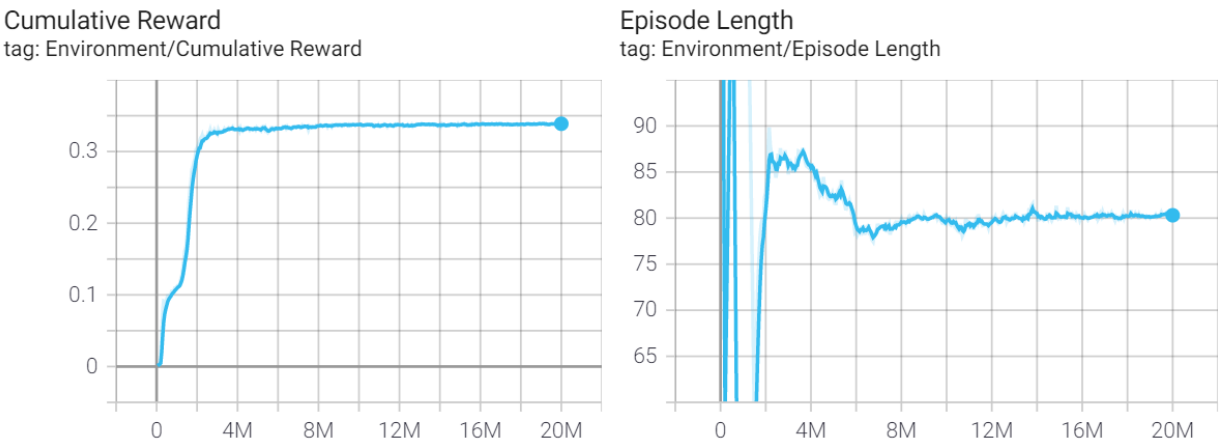

Figura 11. *Gráficas del escenario de Plataformas y movimiento*

Para esta primera fase, se ha realizado un entrenamiento de 20 millones de pasos, el máximo posible. El tiempo de entrenamiento ha sido de 12 horas y 40 minutos, mucho mayor que el anterior. Analizando la gráfica de recompensa, se puede apreciar el punto en el que los agentes consiguen saltar la primera plataforma. Este se corresponde con el tramo de menor pendiente al inicio. El motivo de esta menor pendiente es que antes de conseguir saltar a esta plataforma los agentes avanzan y caen, lo cual es penalizado. Esto ocurre aproximadamente tras 1.5 millones de pasos (25 minutos).

En cuanto a la gráfica de tiempo, para interpretarla es conveniente verla ampliada y con el suavizado al mínimo:

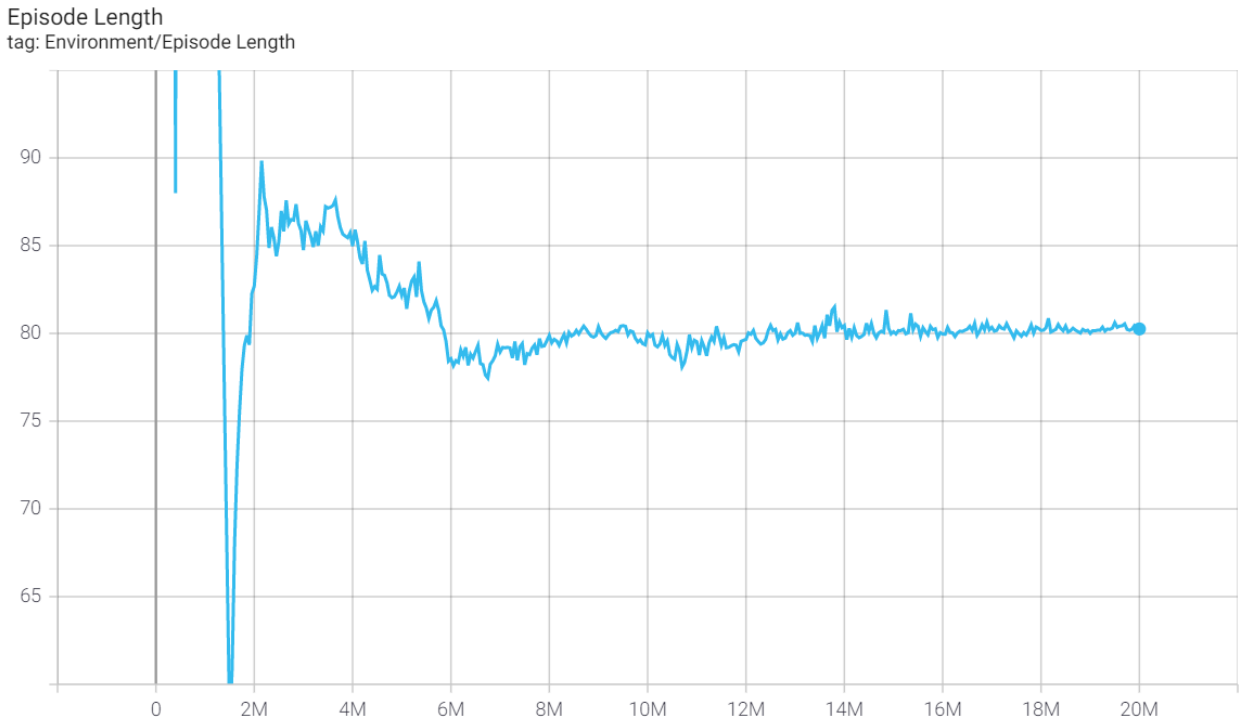

Figura 12. *Gráfica de duración del episodio ampliado y sin suavizado*

En torno a los 1.5 millones de pasos el tiempo desciende bruscamente, debido a que, como se observaba en la gráfica de recompensa, es en este punto cuando los agentes aprenden a avanzar hacia el objetivo, aun a costa de caer. Cuando un agente cae, su episodio termina prematuramente, lo que explica el comportamiento representado.

Aunque en este caso la necesidad de perfilar el modelo era menor, se optó por realizar otra sesión de entrenamiento:

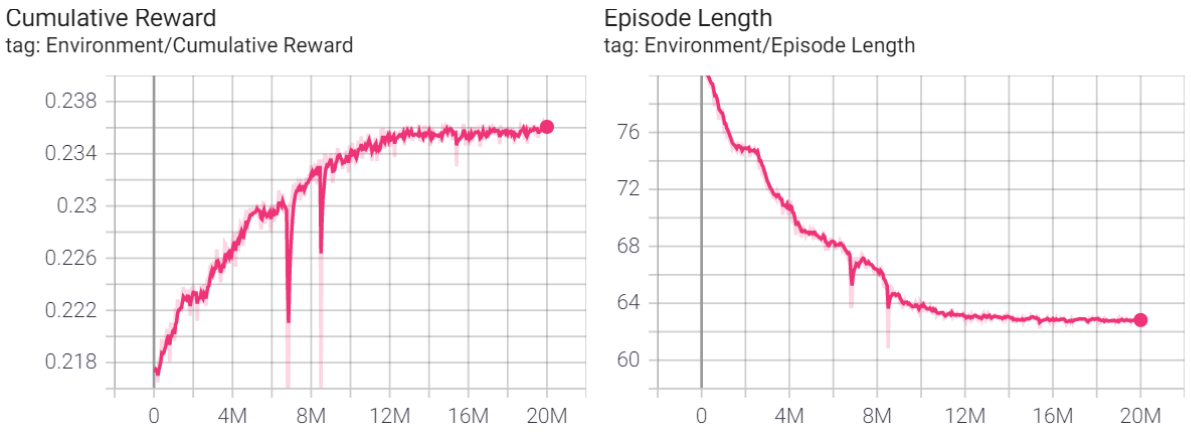

Figura 13. *Gráficas del segundo entrenamiento en el escenario de plataformas*

La principal mejora tras esta segunda fase fue en la plataforma móvil, donde los agentes

aprendieron a elegir el camino más adecuado en función de la posición de la plataforma en lugar de intentar siempre el mismo.

4.1.3 Escenario Precisión y Espera

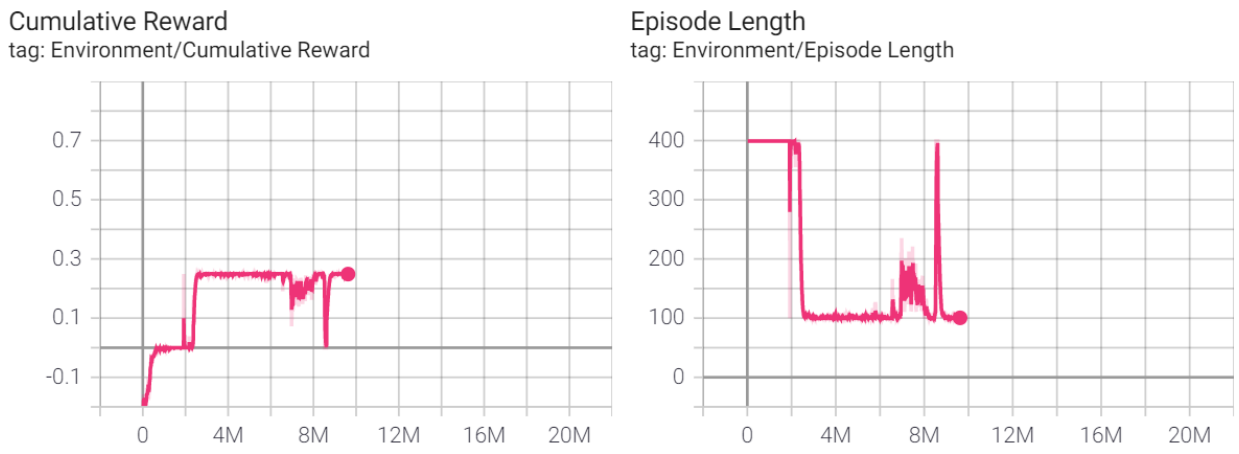

Figura 14. *Gráfica parcial del escenario de Precisión y Espera*

En este caso, el entrenamiento tuvo lugar en 20 millones de pasos (en la gráfica solo se muestran los primeros 9), y el tiempo total fue de 8 horas. Durante la mayor parte del tiempo de entrenamiento, los agentes no eran capaces de subir consistentemente a la plataforma móvil. No fue hasta los últimos momentos que pudieron hacerlo.

4.1.4 Agente Visual

Para el agente visual, tras algunos ajustes, se consiguió que resolviera el escenario de Precisión y Espera , que fue el que se eligió para entrenarlo. Debido a la mayor carga computacional de estos agentes, no fue posible usar más de tres simultáneamente. En 3 millones de pasos y unas 8 horas, el agente estaba correctamente entrenado. Nótese que para que el agente pudiera interpretar el escenario más fácilmente, se usaron colores con mayor contraste para las distintas plataformas.

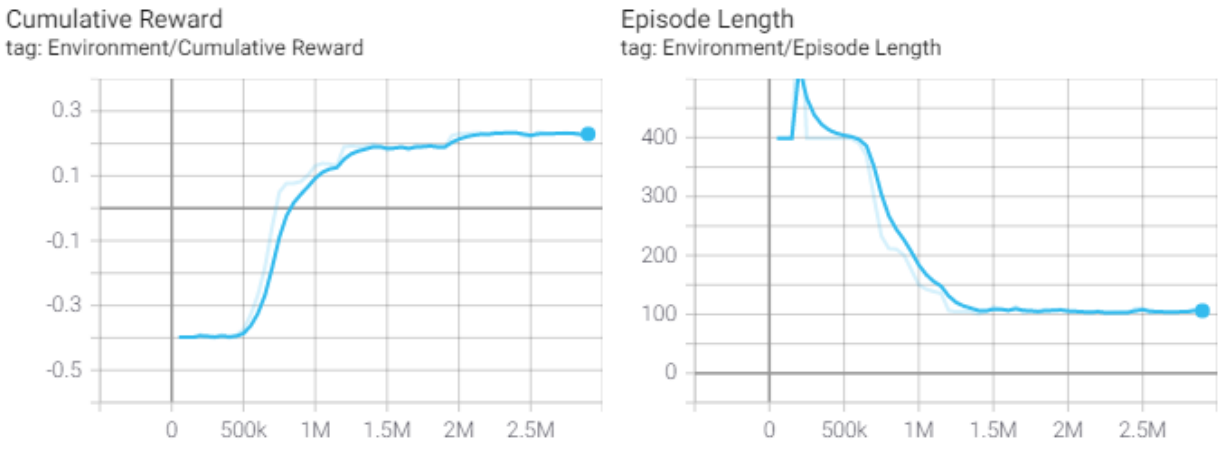

Figura 15. *Gráficas agente visual*

4.1.5 Curiosidad

Las pruebas con el módulo de curiosidad han sido las únicas que no han tenido éxito. Tras dos sesiones de 20 millones de pasos y 18 horas cada una, los agentes no pudieron aprender a completar el escenario. En vista de los resultados, parece que las recompensas extrínsecas están demasiado dispersas, y las recompensas por curiosidad no fomentaron que el agente aprendiera patrones de comportamiento correctos.

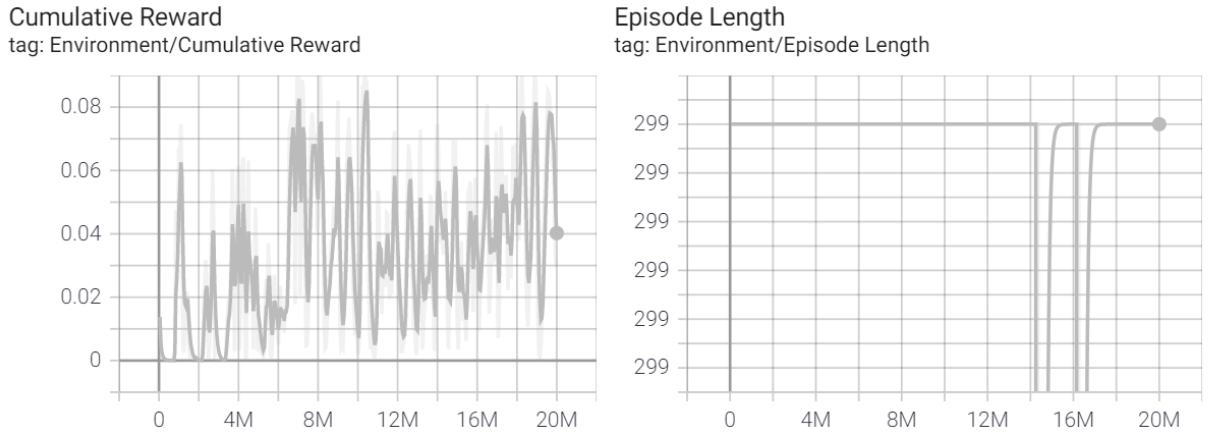

Figura 16. *Gráficas agentes de curiosidad*

Como se puede ver, aunque parece que la recompensa tiene una tendencia ascendente, no llega a converger y la estabilidad es nula.

4.2 Análisis de resultados

En el punto anterior se han presentado los tiempos de entrenamiento, y se ha podido comprobar que el tiempo requerido para entrenar a los agentes no es demasiado alto. Además hay que tener en cuenta que estas pruebas se han hecho con no más de 20 agentes, ejecutándose en un único ordenador y usando la CPU (modelo AMD RYZEN 5 3600). Incrementando el número de agentes y de equipos ejecutando los entrenamientos y, en menor medida, modificando el entrenamiento para que use una GPU, se podrían obtener tiempos menores y entrenamientos más complejos.

Una vez correctamente entrenados, se puede pasar a evaluar la utilidad de los agentes a la hora de testear un videojuego:

4.2.1 Cobertura del escenario

El primer punto a tener en cuenta es la cobertura que ofrece este método. Los agentes exploran el entorno sin preconcepciones de este, por lo que es más probable que

alcancen un mayor número de estados.

Para determinar la cobertura, se pueden generar diagramas de las trayectorias de los agentes como el siguiente:

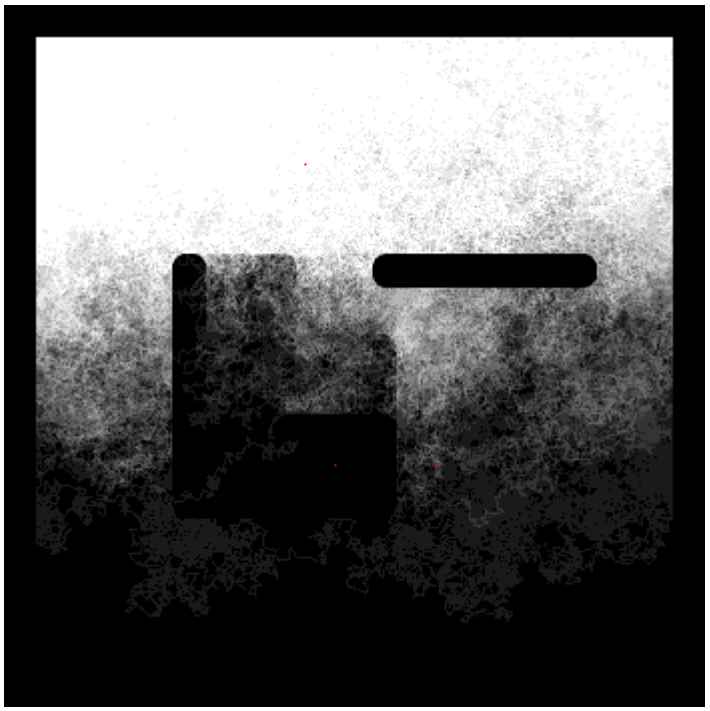

Figura 17. *Muestra de cobertura de la exploración*

Como se puede ver, el espacio explorado es mucho mayor que el que se alcanzaría mediante el uso de scripts. También en este diagrama se puede ver cómo los agentes han encontrado el muro sin colisiones y pasado a través de este.

En cuanto a la cobertura, en las pruebas del Módulo de Curiosidad se consiguió una cobertura más distribuida por el escenario. Si se consiguiera utilizar esta técnica podría resultar muy útil para las pruebas. Una posible solución sería utilizarla junto con la técnica de Reward Shaping usada en el resto de pruebas en lugar de sustituirla, o premiar la exploración directamente a través de la función de recompensa de alguna forma, cambiando dinámicamente entre fomentar la exploración y el alcanzar objetivos.

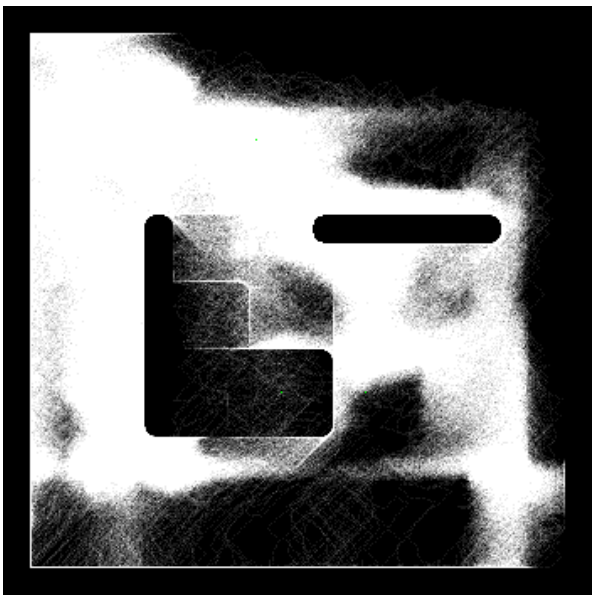

Figura 18. *Cobertura del agente con Módulo de Curiosidad*

4.2.2 Detección de fallos

En cuanto a la detección de fallos en el videojuego (el principal punto de estudio de este trabajo) como se ha visto en el punto anterior, los agentes han conseguido encontrar el muro sin colisiones. Para visualizarlo se puede hacer uso tanto de los diagramas presentados, como de grabaciones de los entrenamientos. En este diagrama (superpuesto sobre una captura del escenario) se puede observar que los agentes han encontrado y utilizado la falta de colisión del muro amarillo para alcanzar los objetivos más rápidamente:

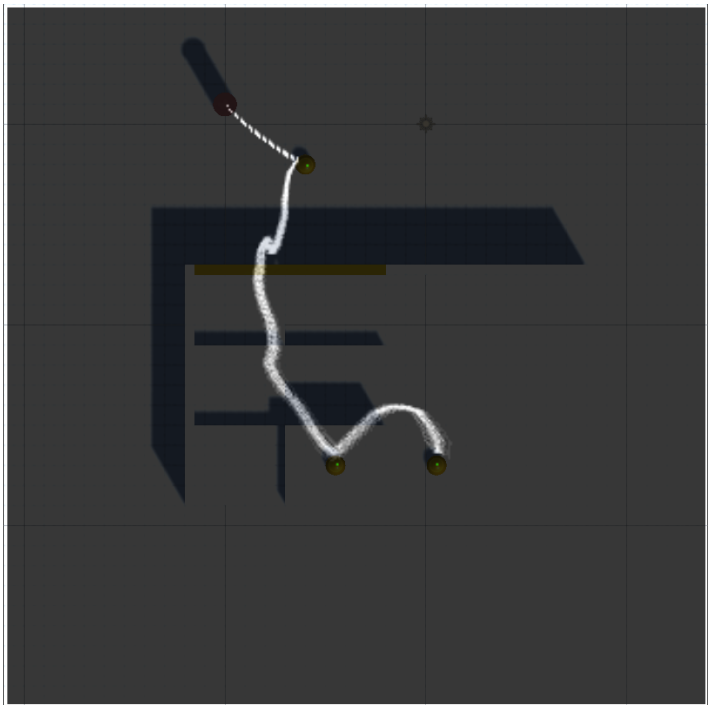

Figura 19. *Muestra de entrenamiento completo*

Además, también se han generado diagramas que permiten encontrar lugares donde los agentes pueden quedar bloqueados. Para esto, se dibujado sobre los diagramas anteriores un punto rojo que indica donde ha terminado el episodio el agente:

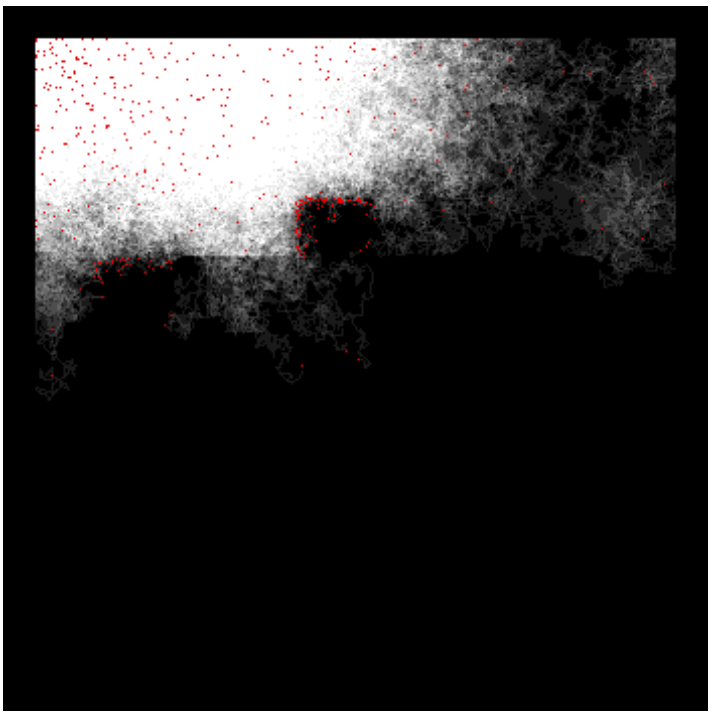

Figura 20. *Diagrama para detectar puntos de bloqueo*

En este ejemplo del escenario de Exploración con zonas de bloqueo, se puede visualizar de forma clara las dos áreas cuadradas donde los agentes quedan bloqueados, que se corresponden a las mayores concentraciones de puntos.

Además, durante las pruebas realizadas, en dos ocasiones los agentes lograron encontrar errores no intencionales. En uno de los escenarios iniciales, los agentes consiguieron alcanzar una recompensa saltando contra la pared debido a un fallo en cómo se detectaba si el agente se encontraba en el suelo. También en el escenario de Precisión y Espera los agentes encontraron una forma de alcanzar la plataforma final directamente desde el suelo, sin tener que subirse a la plataforma móvil. En este caso el error fue la colisión con dicha plataforma.

4.2.3 Evaluación de la dificultad

Adicionalmente, se pueden utilizar los resultados de los entrenamientos para evaluar la dificultad del escenario completo o partes específicas de este. La métrica más sencilla para evaluar la dificultad de un escenario es valorar el tiempo de entrenamiento para alcanzar un cierto nivel de recompensa.

En cuanto a la evaluación de la dificultad de las distintas partes de un escenario, esto se puede hacer mediante dos herramientas mostradas en los puntos anteriores. Por un lado, analizando las gráficas de recompensa se puede encontrar zonas de poca pendiente que se corresponden a momentos en que los agentes han avanzado a un ritmo más lento. Un ejemplo de esto es la gráfica del escenario de Precisión y Espera:

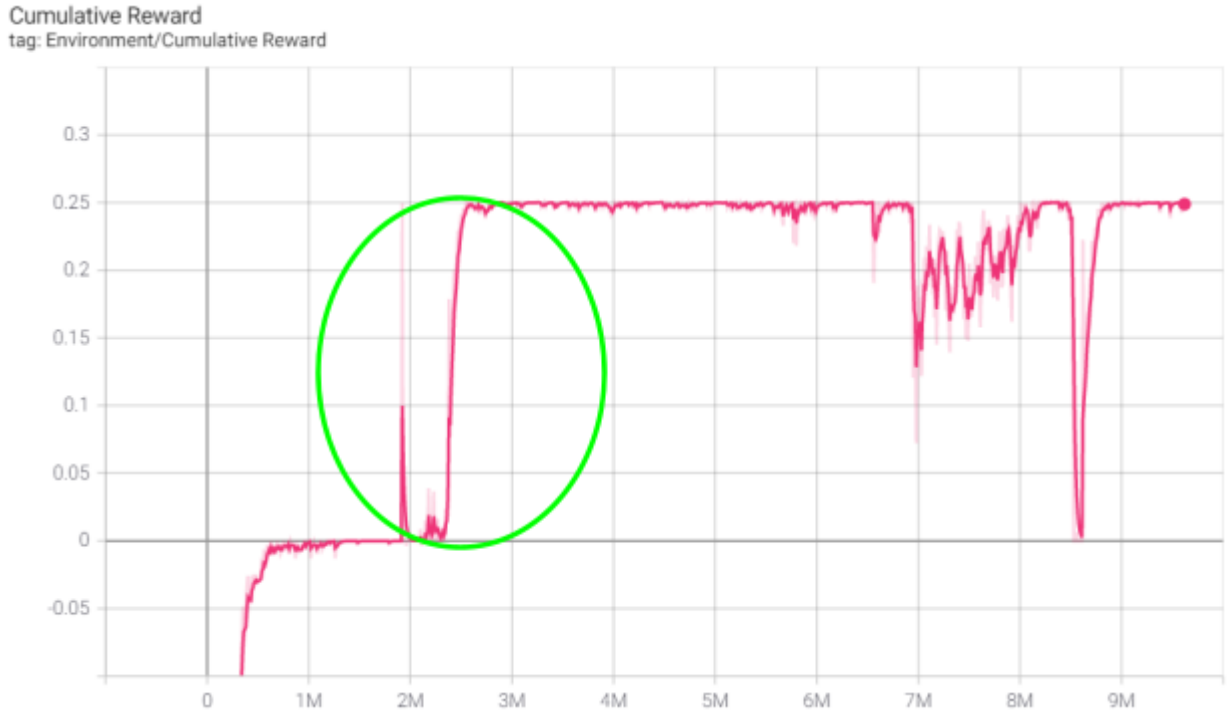

Figura 21. *Gráfica mostrando punto con mayor dificultad*

El punto señalado con el círculo verde se corresponde con el momento en que los agentes aprendieron a subir a la plataforma. Sin embargo, antes de conseguirlo existe un tramo de aproximadamente 1.6 millones de pasos en los que la recompensa aumenta muy lentamente. Con esto, se puede concluir que, al menos para el modelo diseñado, este fue uno de los puntos de mayor dificultad.

Además de analizar la gráfica de recompensa acumulada, también se puede usar un diagrama que muestre los puntos en que los agentes terminan su episodio. Continuando con el ejemplo del escenario Precisión y Espera:

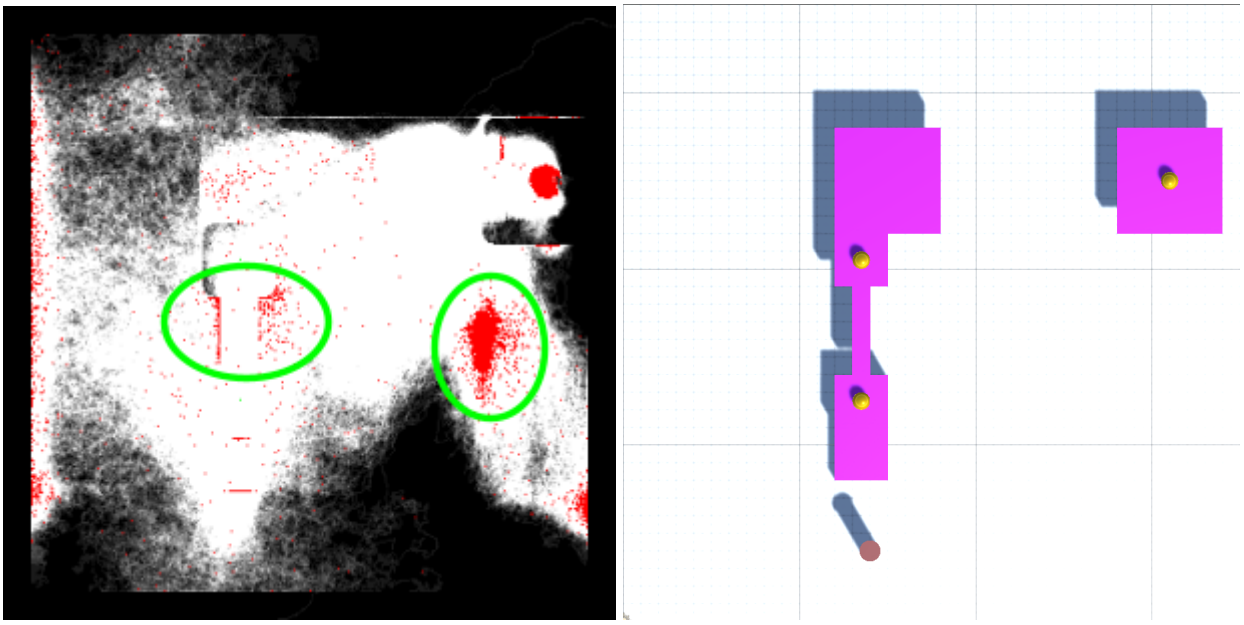

Figura 22. *Diagrama para visualizar zonas de mayor dificultad*

En este caso hay dos concentraciones de puntos que son relevantes. En torno a la pasarela estrecha que los agentes deben atravesar, debido a agentes que caen de ésta, y los de la derecha, que se corresponden con un punto en el que los agentes intentaban acceder a la plataforma final desde el suelo, pero no lo conseguían. Con esto se puede deducir que los dos puntos de mayor dificultad son la pasarela y alcanzar la plataforma final.

Capítulo 5 Conclusiones y líneas futuras

5.1 Conclusiones

Como se ha podido comprobar en la sección de resultados, la mayoría de pruebas realizadas han concluido de manera satisfactoria. Se han conseguido desarrollar tres escenarios con distintas finalidades, cumpliendo así con el primer objetivo específico planteado en la sección 1.2.2.

Tras varias pruebas y la exposición de los agentes a los distintos escenarios se ha podido desarrollar un modelo genérico que es capaz de resolver los tres escenarios planteados como se expuso en el apartado 3.3, tanto en cuanto a los sensores utilizados como a la función de recompensa, con lo que se han cumplido los objetivos específicos segundo y tercero.

En el Capítulo 4 se han presentado los resultados del desarrollo, acompañados por distintos diagramas y un análisis de la utilidad de cada uno. Con esto queda cumplido el cuarto objetivo específico.

Por último, como se introdujo en el apartado 3.5 se han realizado dos pruebas adicionales: la primera usando un agente visual y la segunda basada en el Módulo de Curiosidad. Aunque sólo ha proporcionado resultados satisfactorios el agente visual, también se ha demostrado el potencial del uso del Módulo de Curiosidad, por lo que se considera cumplido el 5 y último objetivo específico planteado.

Habiendo completado todos los objetivos específicos planteados al inicio, se puede concluir que el objetivo general del Trabajo de Fin de Grado se cumple de forma satisfactoria.

5.2 Líneas futuras

Tras completar el desarrollo, hay algunos puntos en los que se podría continuar expandiendo este trabajo. A continuación se enumeran algunos de ellos:

- Diseñar agentes con mecánicas de movimiento más complejas, como doble salto, escalada o deslizamiento.
- Probar escenarios más complejos, potencialmente basados en videojuegos ya existentes, y comprobar la efectividad de este método.
- Realizar pruebas en entornos distribuidos con el fin de realizar entrenamientos en entornos más complejos.
- Expandir y refinar el uso del Módulo de Curiosidad, pues este plantea oportunidades muy interesantes si se pretende fomentar la exploración libre del agente como se ha demostrado en otros trabajos [10].

- Intentar combinar el concepto del agente visual desarrollado con métodos de detección de fallos visuales en videojuegos [12], consiguiendo a la vez encontrar fallos gráficos y en la exploración.

Capítulo 6 Summary and Conclusions

6.1 Conclusions

As it has been shown in the results section, most of the tests carried out have concluded satisfactorily. Three scenarios have been developed with different purposes, thus fulfilling the first specific objective set out in Chapter 1.2.2.

After several tests and the exposure of the agents to the different scenarios, it has been possible to develop a generic model that is capable of solving the three scenarios proposed as explained in Chapter 3.3, both in terms of the sensors used and the reward function. With this, the second and third specific objectives have been met.

In Chapter 4 the development results have been presented, accompanied by different diagrams and an analysis of the usefulness of each one. With this, the fourth specific objective is fulfilled.

Finally, as introduced in Chapter 3.5, two additional tests have been carried out: the first using a visual agent and the second based on the Curiosity Module. Although only the visual agent has provided satisfactory results, the potential of using the Curiosity Module has also been proved, which is why the 5th and last specific objective is considered fulfilled.

Having completed all the specific objectives set out at the beginning, it can be concluded that the general objective of this project is satisfactorily met.

6.2 Future lines

After development is complete, there are some points where this work could be expanded further. Some of them are listed below:

- Design agents with more complex movement mechanics, such as double jumping, climbing, or dashing.
- Test more complex scenarios, potentially based on existing video games, and check the effectiveness of this method.
- Test in distributed environments in order to train in more complex environments.
- Expand and refine the use of the Curiosity Module, as it presents very interesting opportunities when trying to promote the free exploration of the agent, as has been demonstrated in other works [10].
- Try to combine the concept of the visual agent developed with methods for detecting visual glitches in video games [12], managing to find graphical glitches and in the exploration at the same time.

Capítulo 7 Presupuesto

En este capítulo se presenta el presupuesto en función de las horas.

7.1 Coste Material

Objeto	Valor (€)
Ordenador	900

7.2 Coste Humano

Tarea	Horas	Valor por unidad	Valor (€)
Horas de Desarrollo	300	20	6000
Horas de Investigación	100	15	1500
Total	400	-	7500

Tabla 7.1**:** Resumen de tipo

Bibliografía

[1] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, Igor Mordatch, “Emergent tool use from multi-agent autocurricula”

[2] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, Joelle Pineau, “An Introduction to Deep Reinforcement Learning”

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, “Proximal Policy Optimization Algorithms”

[4] Toftedahl Marcus, “A Taxonomy of Game Engines and the Tools that Drive the Industry”

[5] Repositorio del formato ONNX, https://github.com/onnx/onnx a día 07/06/2021

[6] Joshua Hare, "Dealing with Sparse Rewards in Reinforcement Learning"

[7] Nirnai Rao, Elie Aljalbout, Axel Sauer, Sami Haddadin, "How to Make Deep RL Work in Practice"

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver *et al*, “Human-level control through deep reinforcement learning”

[9] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, Trevor Darrell, “Curiosity-driven Exploration by Self-supervised Prediction”

[10] Camilo Gordillo, Joakim Bergdahl, Konrad Tollmar, Linus Gisslén, “Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents”

[11] Carlos García Ling, Konrad Tollmar, Linus Gisslén,

“Using Deep Convolutional Neural Networks to Detect Rendered Glitches in Video Games”

[12] Joakim Bergdahl, Camilo Gordillo, Konrad Tollmar, Linus Gisslén, “Augmenting Automated Game Testing with Deep Reinforcement Learning”