Universidad Politécnica de Cartagena

Evaluación de algoritmos de aprendizaje por refuerzo en un entorno Unity

TRABAJO FIN DE GRADO

Autor:

Cristian Saura Moreno

Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicaciones

Grado en Ingeniería Telemática

Agradecimientos

En especial a mi familia y a todos mis seres queridos por apoyarme durante toda mi etapa académica y finalmente poder formarme como ingeniero. A mis compañeros más allegados, por hacerme sentir a gusto durante toda la carrera, ya que hemos conseguido formar un gran equipo.

Y finalmente a mis profesores ya que sin ellos habría sido imposible todo esto.

Índice

1. Introducción

El uso de la inteligencia artificial en diferentes proyectos, hoy en día, es una necesidad, debido a que gran cantidad de los problemas actuales deben resolver tareas muy complejas para ser solucionadas por una o varias personas. Es por eso que la inteligencia artificial se utiliza en campos muy diversos entre los que se incluyen la economía, administración, medicina, videojuegos, astronomía, entre otras disciplinas. Sin la aplicación de estas técnicas tendrían que recurrir a procesos muy costosos o sin garantías de obtener una solución viable.

En la última década, el aprendizaje por refuerzo ha logrado avances significativos en la búsqueda de lograr inteligencia de propósito [Ward et al., 2020]. En la búsqueda de la inteligencia artificial general, nuestra medida de progreso más significativa es la capacidad de un agente para lograr objetivos en una amplia gama de entornos. Las plataformas existentes para construir tales entornos suelen estar limitadas por las tecnologías en las que se basan y, por lo tanto, solo pueden proporcionar un subconjunto de escenarios necesarios para evaluar el progreso.

Este trabajo fin de grado plantea el diseño de entornos que permitan el desarrollo y la evaluación de algoritmos de inteligencia artificial. En este trabajo se ha optado por desarrollar entornos de tipo videojuego, en línea con una tendencia predominante en este ámbito de investigación desde que Deepmind convirtió los juegos clásicos de Atari en un banco de pruebas estándar para inteligencia artificial en su publicación de 2015 [Mnih et al., 2015]. Cabe destacar que el sector de los videojuegos tiene más de medio siglo de vida, pero es ahora cuando los videojuegos están despuntando más que nunca. Esto se debe a que actualmente la gran mayoría de la población cuenta con mucha más tecnología, ya sea directamente con un smartphone o con cualquier otra herramienta como podría ser un ordenador, además de que actualmente los videojuegos son más accesibles y existe mayor diversidad de plataformas para el desarrollo de los mismos.

Para la realización de este proyecto, haremos uso de las técnicas de aprendizaje por refuerzo, con la finalidad de hacer que nuestro agente aprenda. La investigación en el área del aprendizaje por refuerzo o reinforcement learning (RL) es muy activa actualmente. Estas técnicas permiten que un agente (algoritmo de control) pueda aprender a partir de su experiencia acumulada durante la interacción con el entorno, premiando comportamientos que acercan al cumplimento de la tarea y castigando los que llevan a fracasar. Se hará uso de un motor gráfico, Unity, para la implementación del entorno virtual donde se desarrollará el videojuego. Unity incluye un kit de herramientas de aprendizaje denominado ML-Agents, esto es un juego de herramientas listo para usar que incluye una variedad de algoritmos de aprendizaje, regímenes de entrenamiento y ejemplos de entornos, así como un mecanismo para conectar agentes definidos externamente [Bolt et al., 2020]. Su kit de herramientas también proporciona una API de Python para permitir que los agentes definidos externamente interactúen con los juegos configurados [Hemmings et al., 2020]. En este proyecto se utilizará este kit de herramientas junto con TensorFlow, que nos permitirán realizar y monitorizar el aprendizaje de agentes RL en nuestro entorno (videojuego). Adicionalmente, se aplicarán los algoritmos y se realizará un seguimiento del estudio en tiempo real. Unity es compatible con las librerías anteriormente mencionadas y proporciona ciertas facilidades en su manejo, además cuenta con librerías propias para incluir elementos que mejoran los resultados de los entrenamientos.

Para el entrenamiento del agente, o personaje del videojuego, se utilizará el algoritmo PPO (Proximal Policy Optimization). El algoritmo de optimización de política próxima (PPO) es un algoritmo de aprendizaje por refuerzo profundo con un rendimiento excepcional, especialmente en tareas de control continuo. Pero el rendimiento de este método todavía se ve afectado por su

capacidad de exploración [Zhang et al., 2020]. Además, se realizarán pruebas con el algoritmo SAC (Soft Actor Critic) que es un algoritmo de aprendizaje por refuerzo de última generación para escenarios de acción continua [Christodoulou et al., 2019] para comprobar si es igual de eficiente que PPO para este tipo de entornos. A partir de ahí realizar los entrenamientos con el que mejor responda.

1.1 Objetivos

El objetivo principal de este proyecto es implementar un entorno de tipo videojuego, para el entrenamiento de agentes mediante el uso de técnicas de aprendizaje por refuerzo.

De este modo, se pretende dar una visión general de cómo se debe empezar a desarrollar un entorno de este tipo haciendo uso de las herramientas anteriormente citadas y cómo ir ajustando el agente para que mejore su desempeño.

Otros objetivos secundarios son, por un lado, aplicar las herramientas TensorFlow y Unity para el desarrollo del proyecto y, por otro lado, mejorar el rendimiento del agente de modo que no valga simplemente con que cumpla su función, sino que se intentará que desempeñe la acción lo mejor posible, ajustando los valores de sus parámetros de configuración.

1.2 Motivación

En el mundo de la tecnología, la inteligencia artificial ha jugado un papel muy importante en distintos mundos, como pueden ser el transporte, la ciencia ficción, los videojuegos, etc. Con la incorporación del aprendizaje de máquinas o machine learning, la tecnología se encuentran , actualmente, en continuo crecimiento y desarrollo, tanto es así, que se ha llegado a desarrollar máquinas que aprenden por sí solas dependiendo de los estímulos externos que reciben.

El aprendizaje de máquinas se puede aplicar para otros usos como puede ser la medicina, donde se podría dotar, por ejemplo, de inteligencia a un instrumento médico de monitorización de pacientes para que reconozca ciertas situaciones y actúe en consecuencia.

Todo esto hace que la inteligencia artificial junto con el aprendizaje de máquinas sean temas a tener en cuenta para multitud de empresas. El uso de estas tecnologías puede tener diversas aplicaciones. Cada día se encuentran nuevos usos, lo que hace que nos demos cuenta de las infinitas posibilidades que tienen.

1.3 Contribuciones

En este proyecto se han comparado dos algoritmos de aprendizaje, SAC y PPO, para el entrenamiento del agente. Se le ha implementado un script al agente con la lógica a seguir y se ha desarrollado un escenario en Unity que consta de 5 plataformas con distintos niveles de dificultad.

El entrenamiento se ha realizado de dos formas distintas, lo que ha permitido obtener diferentes conclusiones. Por otra parte, los parámetros de configuración del entrenamiento han sido modificados hasta encontrar unos parámetros de los que se obtuvieran buenos resultados y partiendo de ellos se han ajustado más durante el desarrollo del proyecto

1.4 Contenidos

El resto del documento se organiza de la siguiente forma. En el Capítulo 3 se describen las tecnologías utilizadas, donde se explican qué herramientas se han utilizado, para qué son y cómo se han usado. El Capítulo 4 presenta, la solución propuesta, donde se muestra qué entorno se ha desarrollado y en qué partes se divide. El Capítulo 5 detalla las pruebas realizadas durante el desarrollo del proyecto y las conclusiones que se obtienen de cada prueba y, finalmente, el Capítulo 6 contiene las conclusiones obtenidas del proyecto realizado.

2. Tecnologías usadas

Para poder crear el entorno, monitorizarlo, entrenarlo y que realice correctamente su función necesitamos hacer uso de un conjunto de herramientas que se desarrollaran en este capítulo.

2.1 Machine Learning

El Machine Learning o aprendizaje de máquinas, es una disciplina perteneciente al campo de la Inteligencia Artificial (IA) que, utilizando algoritmos, permite dotar a ordenadores de la capacidad de identificar patrones en conjuntos muy elevados de datos y, en consecuencia, hacer predicciones. Este aprendizaje consigue que los ordenadores realicen tareas específicas de forma autónoma. Dentro de Machine Learning, el aprendizaje se puede dividir en tres categorías que son aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo. Este proyecto se focalizará en el aprendizaje por refuerzo.

El aprendizaje por refuerzo, en este caso, permitirá que un agente pueda aprender de sus errores, mediante un sistema de recompensas, y de esta forma tome la mejor decisión ante diferentes situaciones.

En la figura 1 se puede observar cómo funciona el aprendizaje por refuerzo. Primero se crea un entorno en el que se debe resolver un problema. Después se introduce un agente para que lo resuelva tomando como punto de partida el estado en que se encuentra. En cada estado se pueden realizar acciones que cambiarán el estado, donde habrá una política que se encargará de elegir las acciones a tomar en función del estado actual. Tras tomar una acción se recibirá una recompensa, que se usará para actualizar la política de forma que favorezca ejecución de acciones que maximicen la recompensa acumulada a lo largo de un episodio (ciclo comprendido desde el inicio de una ejecución de la tarea hasta su fin). El ciclo se repetirá hasta que termine el entrenamiento y finalmente el agente aprenderá a realizar su función lo mejor posible, dependiendo de los parámetros de configuración establecidos y del algoritmo de aprendizaje utilizado.

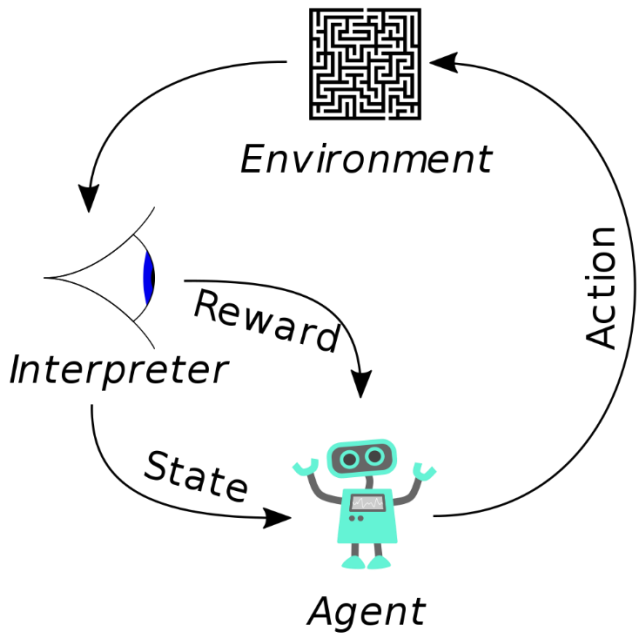

Figura 1: Aprendizaje por refuerzo

Dentro del aprendizaje por refuerzo se pueden utilizar distintos algoritmos de entrenamiento. En este proyecto nos centraremos, principalmente, en PPO, aunque también se entrenará en un mismo entorno el algoritmo SAC y PPO para comprobar cual de los dos es mejor para este tipo de problema.

2.1.1 PPO

Proximal Policy Optimization (PPO) es un popular algoritmo de aprendizaje profundo (o Deep learning) basado en políticas de gradiente. En las implementaciones estándar, PPO regulariza las actualizaciones de políticas con índices de probabilidad recortados y parametriza las políticas con distribuciones gaussianas continuas o distribuciones Softmax discretas. Estas opciones de diseño son ampliamente aceptadas y están motivadas por comparaciones empíricas de desempeño en los puntos de referencia de MuJoCo y Atari[Christodoulou et al., 2019].

PPO es el algoritmo de aprendizaje que vamos a utilizar para que el agente aprenda a realizar correctamente la tarea. Para usar correctamente este algoritmo es necesario configurar una serie de parámetros que nos permitirán ir ajustando el entrenamiento para conseguir los mejores resultados posibles.

Parámetro	Rango de valores	Descripción
Buffer Size	2048-409600	corresponde a cuántas experiencias (observaciones de agentes, acciones y recompensas obtenidas) deben recopilarse antes de realizar cualquier aprendizaje o actualización del modelo. Normalmente, un mayor buffer_size corresponde a actualizaciones de entrenamiento más estables.
Batch Size	512-5120	es el número de experiencias utilizadas para una iteración de una actualización de descenso de gradiente. Si se está utilizando un espacio de acción continuo, este valor debe ser grande (del orden de 1000)
Learning Rate	$1e^{-5}$-$1e^{-3}$	corresponde a la fuerza de cada paso de actualización de descenso de gradiente. Por lo general, esto debe reducirse si el entrenamiento es inestable y la recompensa no aumenta constantemente.
Beta	$1e^{-4}$-$1e^{-2}$	corresponde a la fuerza de la regularización de la entropía, lo que hace que la política sea "más aleatoria". Esto asegura que los agentes exploren adecuadamente el espacio de acción durante el

		entrenamiento. Incrementar esto asegurará que se tomen más acciones aleatorias. Esto debe ajustarse de manera que la entropía (medible desde TensorBoard) disminuya lentamente junto con los aumentos en la recompensa.
Epsilon	0.1-0.3	corresponde al umbral aceptable de divergencia entre las políticas antiguas y nuevas durante la actualización del descenso de gradientes.

Tabla 1: Parámetros de configuración a tener en cuenta

2.1.2 SAC

Soft Actor Critic (SAC), es un algoritmo de aprendizaje enfocado para tareas robóticas del mundo real. Unos mismos parámetros de configuración pueden valer para muchos entornos, esto hace que SAC sea una apuesta segura a la hora del desarrollo robótico. Aunque en este proyecto nos focalizamos en el campo de los videojuegos, se usará para comprobar que el algoritmo PPO es más eficiente para el trabajo propuesto.

2.1.3 Anaconda navigator

Es un software, de código abierto, que nos permite manejar entornos separados para diferentes distribuciones de Python. Además, nos permite instalar dentro de este entorno la librería de TensorFlow que utilizamos para entrenar a los agentes con diferentes algoritmos de machine learning.

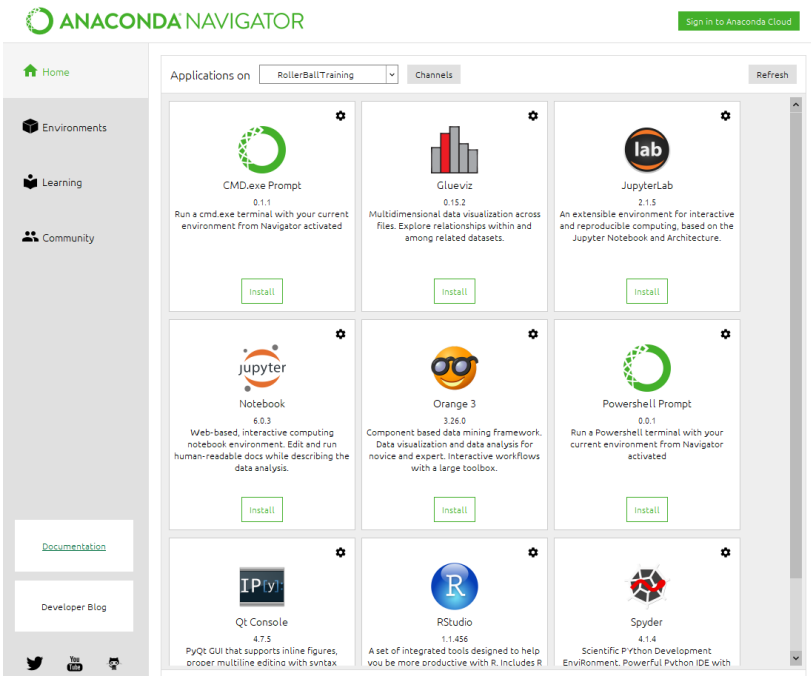

Figura 2: Interfaz de Anaconda

2.2 Unity

Unity es una herramienta diseñada para el desarrollo de videojuegos creada por la empresa Unity Technologies. Unity no engloba únicamente motores para el renderizado de imágenes, de físicas de 2D/3D, de audio, de animaciones y otros motores, sino que engloba además herramientas de detección de elementos (Raycast), herramientas de navegación NavMesh para Inteligencia Artificial o soporte de Realidad Virtual. En nuestro caso nos enfocaremos en físicas de 3D e Inteligencia Artificial.

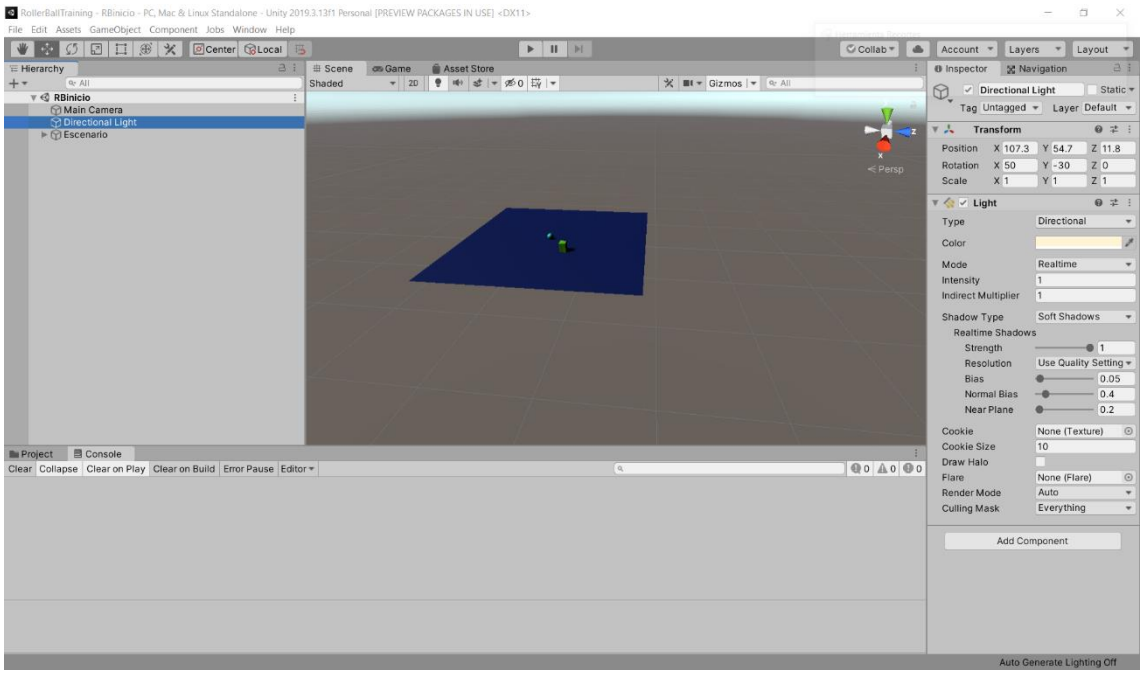

Figura 3: Interfaz de Unity

En este proyecto se utilizará Unity para crear el entorno virtual que queremos entrenar con ML-Agents. Unity dispone de diversas librerías integradas que permiten crear fácilmente figuras geométricas y modificar sus dimensiones, adicionalmente se puede dotar de inteligencia a cada figura. Además, cualquier cambio que se hagan lo podremos apreciar, en la ventana de escena, una vez ejecutado. Todo esto hace a Unity una herramienta fundamental en el desarrollo de entornos virtuales.

2.2.1 ML-Agents

Es una librería de Unity, que permite que los juegos y simulaciones sirvan como entornos para entrenar agentes inteligentes. Esta librería cuenta con diferentes scripts de Python que permiten capacitar fácilmente a agentes de inteligencia. ML-Agents cuenta con diversos elementos para hacer que el aprendizaje del agente sea más rápido, entre ellos, cabe destacar el Raycast, del cual se hace uso en este proyecto. Además, ML-Agents cuenta con sus propios entornos de ejemplos ya entrenados.

Para llevar a cabo un entrenamiento en Unity, primero se debe crear un proyecto en Unity hub, abrir un terminal de Anaconda, seleccionar el proyecto anteriormente creado e introducir el siguiente comando:

Mlagents-learn config/ppo/rollerball_config.yaml –train –run-id=”RollerBall”

Y para continuar el entrenamiento:

Mlagents-learn config/ppo/rollerball_config.yaml –train –initialize-from=”RollerBall” –force

2.2.1.1 Raycast

Raycast es una técnica que se basa en crear imaginarios rayos que parten del personaje en una determinada dirección y ver con qué otros objetos colisionan. De esta forma, puede ayudar a un agente a reconocer el entorno. En el caso particular de ML-Agents, se puede usar un componente llamado RayPerceptionSensor que lance más de un rayo. Como se puede ver en la figura 5, se puede programar para que los rayos informen la posición de objetos con unas determinadas etiquetas, es decir, permitirá detectar elementos del escenario e identificarlos a una cierta distancia. En este proyecto se utilizará para detectar qué tipo de elemento tiene el agente en frente de él y, de este modo, actuar en consecuencia. El elemento Ray Perception Sensor, en la interfaz de Unity, resultará muy útil para que el agente desempeñe mejor su tarea, y por lo tanto mejorará la eficiencia del entrenamiento.

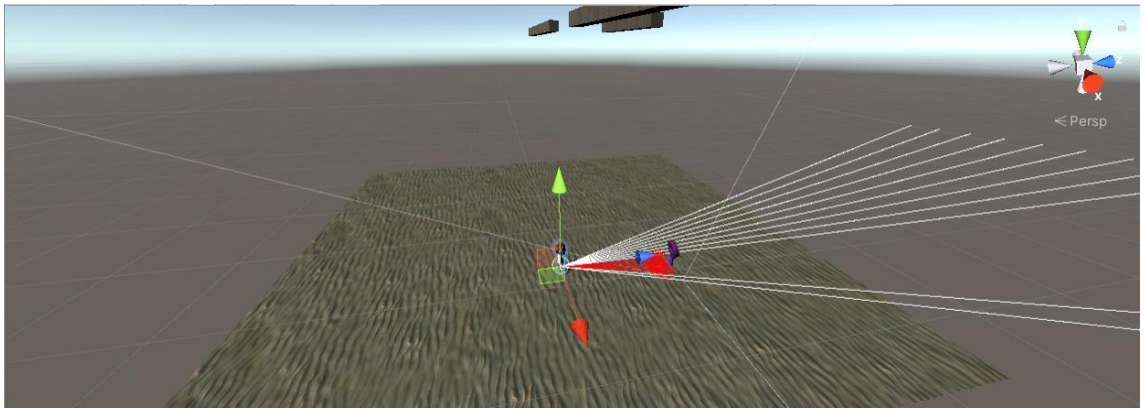

Figura 4: Raycast en Unity

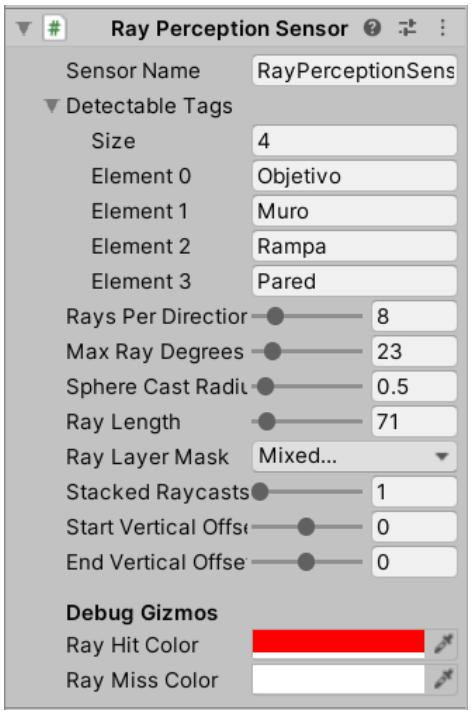

Figura 5: Vista del elemento Raycast en la interfaz de Unity.

Como se puede apreciar en la imagen (figura 5), el elemento Ray Perception Sensor tiene diversos parámetros. Estos parámetros se utilizan para detectar ciertos elementos del escenario al igual que el número y longitud de Raycast con los que va a contar el agente, entre otros.

2.3 TensorBoard

Con esta herramienta podemos monitorizar el entrenamiento. Nos facilita gráficas importantes, como la de recompensa, la de tiempo medio en terminar el episodio y la entropía entre otras, que nos da una idea de cómo de exitoso ha sido el entrenamiento.

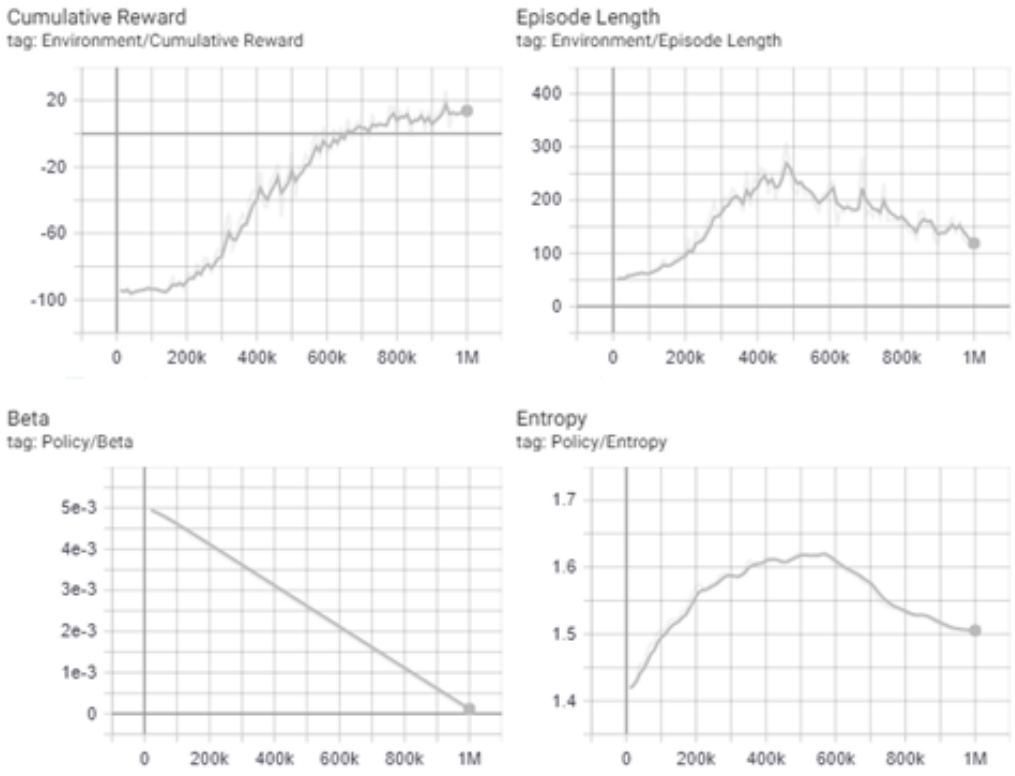

Figura 6: Ejemplo de monitorización de entrenamiento en TensorBoard

Como se puede observar en la imagen (figura 6), cada gráfica cuenta con sus propios valores de coordenadas. En este proyecto se trabajará con estas cuatro gráficas, las cuales representan la recompensa acumulada, el tiempo medio en terminar cada episodio, el valor de beta durante el entrenamiento para representar la aleatoriedad de las políticas escogidas y finalmente la entropía.

En todas estas gráficas el eje X representa los pasos de los que consta el entrenamiento, los cuales se indican en los parámetros de configuración.

3. Solución propuesta

En este capítulo se verá el punto de partida del proyecto y su evolución hasta conseguir la versión final del mismo. Para ello, se utilizará la herramienta Unity con la finalidad de diseñar el escenario en el que se entrenará al agente. Además, se explicará, de forma detallada, las partes de la implementación más importantes, para el funcionamiento de la lógica del agente. En la versión final del proyecto se presentará un entorno formado por cinco plataformas, en las cuales se va incrementando la dificultad, que deben ser superadas por el agente sin caer.

3.1. Elementos básicos del entorno

En este proyecto como punto de partida, se partirá de un escenario muy simple, con el objetivo de comprobar cuál de los dos algoritmos es más eficiente. Este escenario contará con tres elementos: la plataforma, el agente y el objetivo. Además, se contará con una observación adicional y unas recompensas para que cumpla correctamente la tarea.

3.1.1 Elementos del escenario

Como ya se ha mencionado, se trabajará con tres elementos:

- Esfera azul: Representa al agente, nuestro personaje del videojuego, es el encargado de realizar la tarea.
- Bloque verde: Representa al objetivo, este elemento no está dotado de cerebro por lo que solo permanece estático hasta que el agente lo toca, y a continuación aparece en otra posición aleatorio de la plataforma.
- Plataforma: Representa el suelo de nuestro escenario, en él deberá desarrollarse la tarea. Siempre que el agente se caiga volverá a aparecer de nuevo en el centro.

3.1.2 Objetivo

En este escenario (ver Figura 7) el agente debe cumplir la tarea de acercarse al objetivo hasta tocarlo y evitar caerse de la plataforma, en el proceso. De esta forma el agente debe aprender a coger el objetivo el máximo número de veces sin caer.

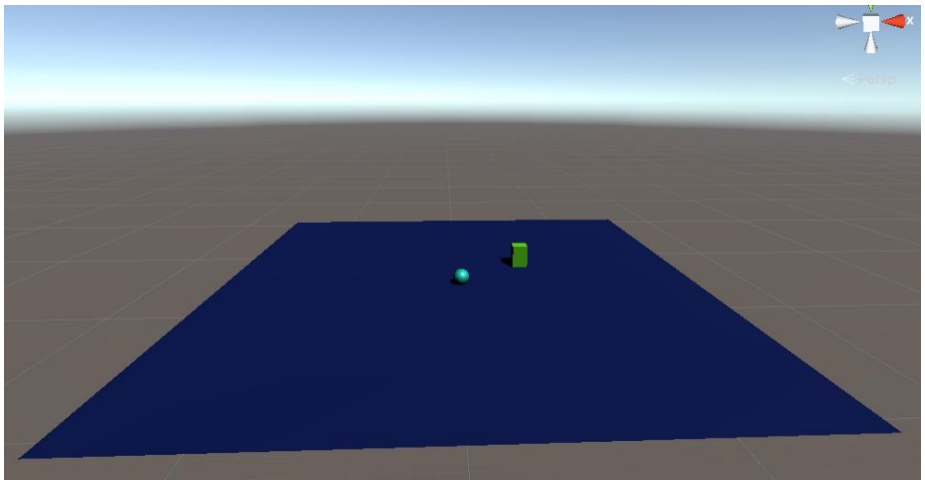

Figura 7: Primer escenario

Como se puede observar, el escenario es muy básico. Esto es así, debido a que inicialmente no se sabía cuál de los dos algoritmos de aprendizaje era más eficiente para este tipo de tareas. Por lo tanto, en vez de complicar el entorno y después probar los dos algoritmos, se propuso este escenario inicial para observar los resultados. En base a eso, más adelante, se explicará qué algoritmo resultó más eficiente y por qué.

3.1.3 Recompensas

Para que el agente aprenda es necesario guiarlo por medio de recompensas, es decir, para que el agente aprenda a realizar la tarea se le tiene que dar algún tipo de estímulo para que sepa si lo está haciendo bien o no. Cada vez que el agente toca al objetivo se le añade una recompensa positiva unitaria. Por otro lado, si el agente cae de la plataforma se le añade una recompensa negativa diez veces mayor que la recompensa positiva. De esta forma se pretende que el agente aprenda correctamente en función de la recompensa acumulada que va obteniendo durante el entrenamiento.

3.1.4 Observaciones adicionales

Para mejorar la eficiencia del entrenamiento, al agente se le añade un Raycast. Este elemento ayuda al agente a recibir estímulos externos más claros, ya que le permite detectar los elementos del escenario. Por lo tanto, se espera que durante el entrenamiento el agente aprenda a reconocer el objetivo y vaya directamente hacia él cada vez que lo vea.

3.1.5 Parámetros de configuración

Para empezar a entrenar al agente con un algoritmo de aprendizaje, es necesario establecer unos parámetros de configuración. En este caso se utilizaron dos algoritmos, anteriormente mencionados, PPO y SAC para comprobar con cual de los dos se obtenían mejores resultados y, en consecuencia, utilizar este algoritmo para los posteriores entrenamientos. Por lo tanto se utilizaron dos parámetros de configuración distintos, uno para el algoritmo PPO (Figura 8) y otro para el algoritmo SAC (Figura 9).

```yaml
behaviors:
    RollerBall:
        trainer_type: ppo
        hyperparameters:
          batch_size: 1000
          buffer_size: 10000
          learning_rate: 3.0e-4
          beta: 5.0e-3
          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: 1000000
        time_horizon: 64
        summary_freq: 10000
```

Figura 8: Parámetros de configuración PPO

```yaml
RollerBall1:
    trainer: sac
    hyperparameters:
        batch_size: 128
        buffer_size: 500000
        buffer_init_steps: 0
        hidden_units: 128
        init_entcoef: 1.0
        learning_rate: 0.0003
        learning_rate_schedule: constant
        max_steps: 500000
        memory_size: 256
        normalize: true
        num_update: 1
        train_interval: 1
        num_layers: 2
        time_horizon: 1000
        sequence_length: 64
        summary_freq: 3000
        tau: 0.005
        use_recurrent: false
        vis_encode_type: simple
        reward_signals:
        extrinsic:
        strength: 1.0
        gamma: 0.99
```

Figura 9: Parámetros de configuración SAC

Para empezar, se entrenó al agente con los dos algoritmos y se observó que con el algoritmo PPO se obtenían unos resultados mejores que con el algoritmo SAC y que además el entrenamiento con SAC era mucho más lento que con PPO. De esta forma se concluyó que el algoritmo PPO era

más eficiente que el SAC para este tipo de tareas y por tanto, se descartó el algoritmo SAC. De aquí en adelante, para continuar el desarrollo del proyecto, solo se trabajará con el algoritmo PPO.

3.2. Entorno definitivo

Una vez entrenado al agente para resolver esa tarea, se diseñó la versión final del proyecto. Esta versión consiste en un videojuego donde un personaje deberá salir de una torre (Figura 13). Para ello, el personaje, representado por un muñeco (Figura 10), deberá encontrar los dos objetivos que habrá en cada plataforma. Estos objetivos no aparecerán de forma simultánea en la plataforma, sino que el personaje primero deberá coger el primer objetivo, representado por una llave (Figura 11), que aparece en una posición aleatoria de la plataforma. Una vez obtenido, se generará, el segundo objetivo en el centro de la plataforma, representado por un cofre (Figura 12). Cuando coja el segundo objetivo el personaje aparecerá en el siguiente piso. De esta manera, el personaje irá escalando pisos hasta llegar al último. El objetivo final es que el personaje consiga terminar la torre sin caerse, es decir, el personaje deberá coger la llave y el cofre en cada uno de los pisos sin caerse. Pero con el fin de que aprenda con mayor rapidez, cada vez que se caiga de la plataforma de un piso, se reseteará el piso en el que se encuentre en vez de empezar directamente desde el primer piso cuando se caiga desde cualquier plataforma.

Figura 10: Personaje

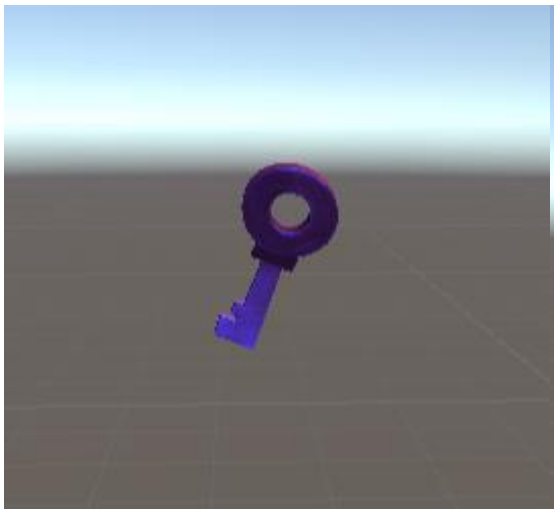

Figura 11: 1º Objetivo

Figura 12: 2º Objetivo

Como se ha mencionado en el apartado anterior, la propuesta final para este proyecto será una torre (Figura 13). Esta torre se compondrá de cinco pisos, en los que la dificultad para encontrar ambos objetivos se irá incrementando a medida que el agente vaya subiendo. A continuación, se presentarán los distintos pisos y se explicará cual es la dificultad a la que el agente se enfrentará en cada uno de ellos.

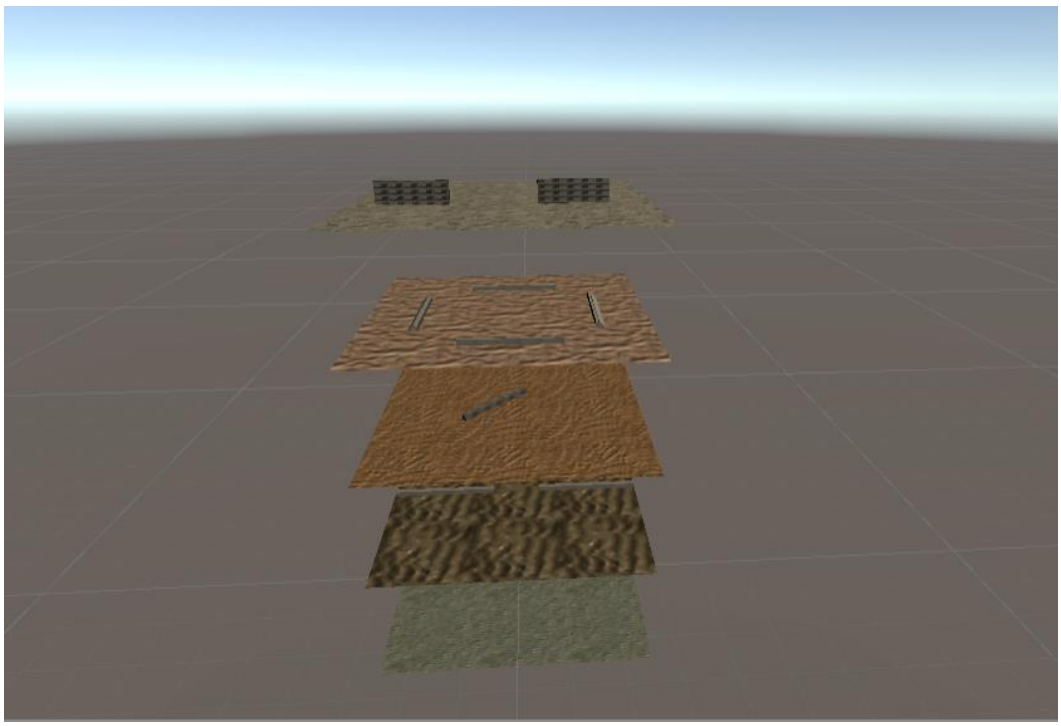

Figura 13: Apariencia de la versión final del juego

3.2.1 Piso 0

Punto de partida del juego, en este piso la única dificultad es no caerse del escenario. En las siguientes imágenes se muestra cómo se vería dentro del juego cada uno de los elementos.

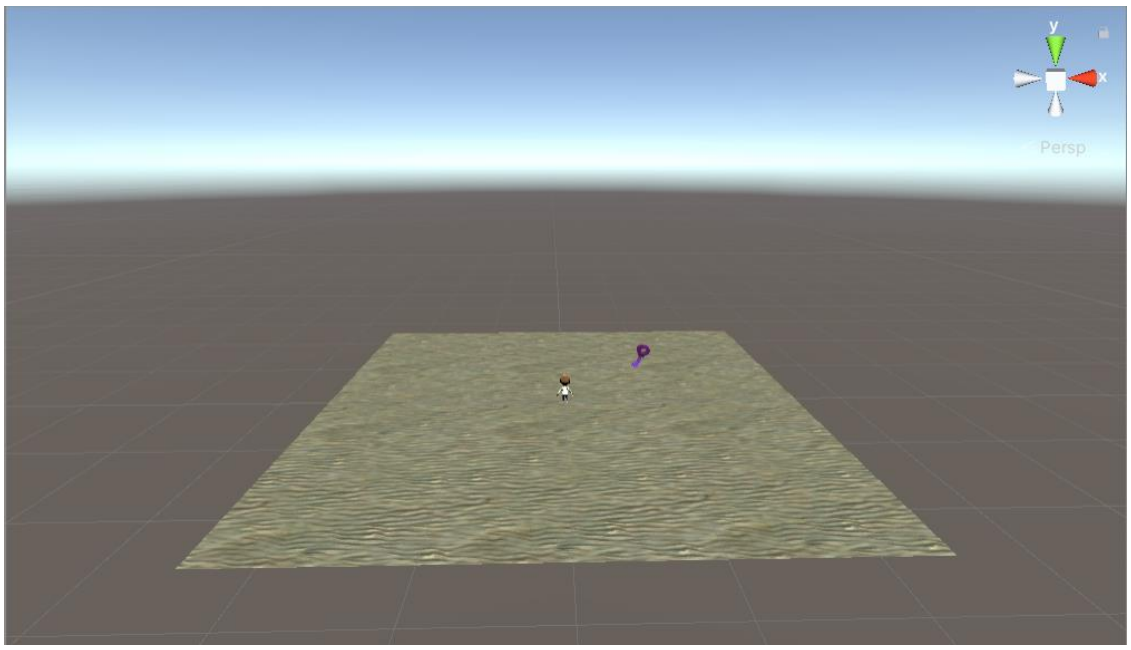

Figura 14: Piso 0 con 1º objetivo (Llave)

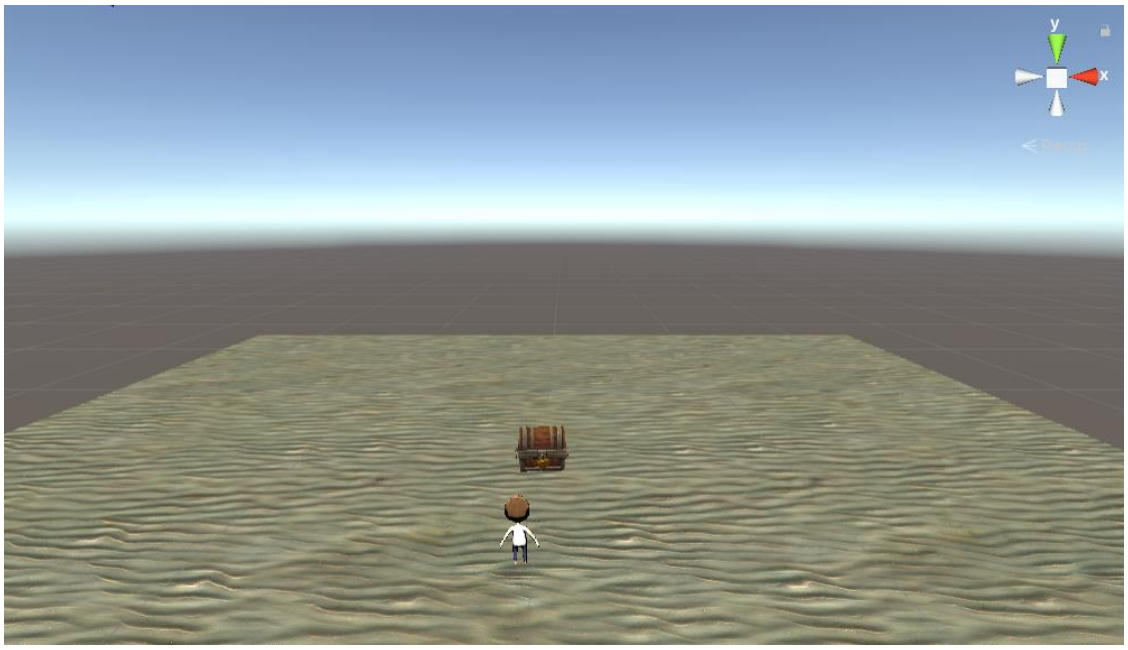

Figura 15: Piso 0 con 2º objetivo (Cofre)

3.2.2 Piso 1

En este piso se le añaden paredes al escenario que permiten activar la opción “saltar” del agente cuando las ve a 5 metros de distancia o menos. Este escenario se diseñó específicamente para que el agente aprendiese a saltar cuando veía una pared.

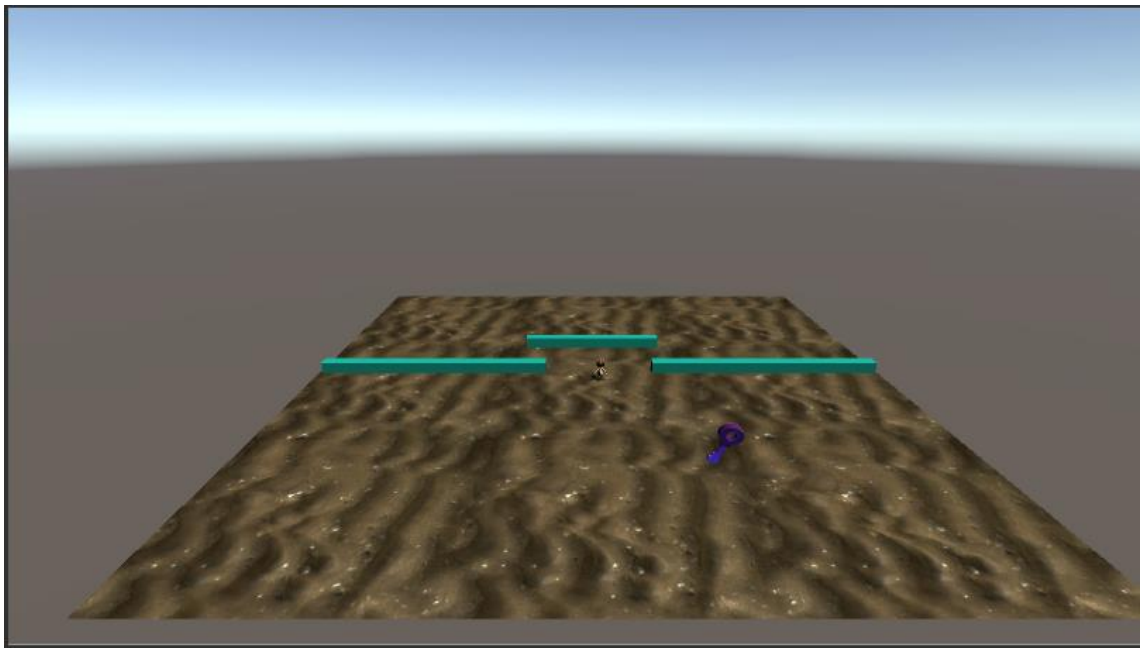

Figura 16: Piso 1 con 1º objetivo (Llave)

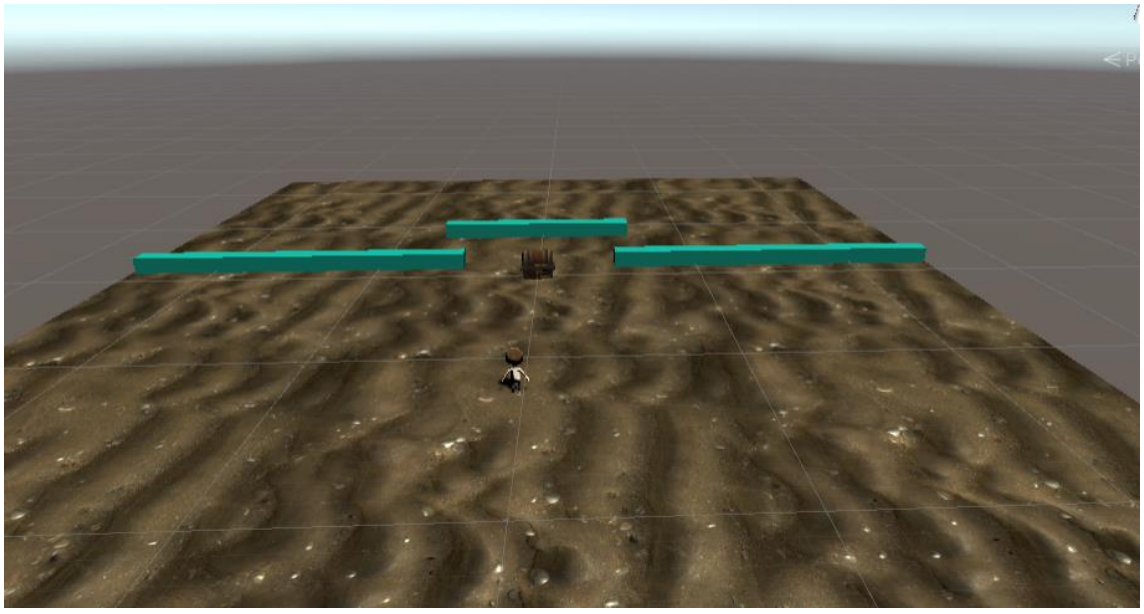

Figura 17: Piso 1 con 2º objetivo (Cofre)

3.2.3 Piso 2

En este piso, la dificultad radica en que se añade un nuevo elemento, un muro, el cual, a diferencia de las paredes, no habilita la opción de saltar y, por lo tanto, el agente, solo puede bordearlo.

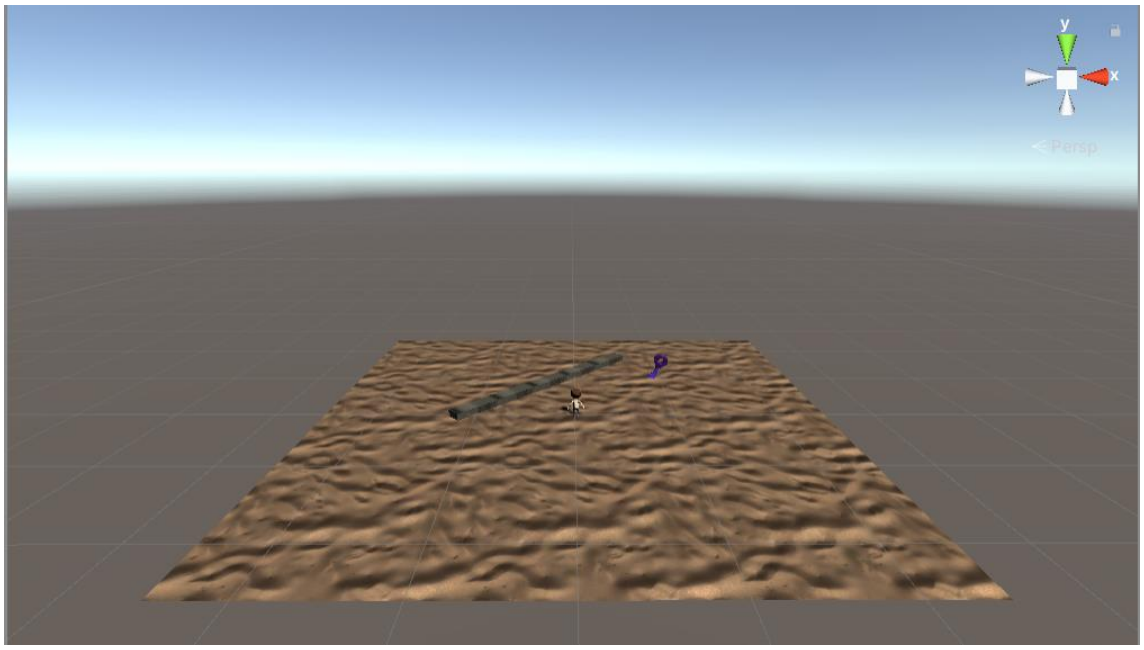

Figura 18: Piso 2 con 1º objetivo (Llave)

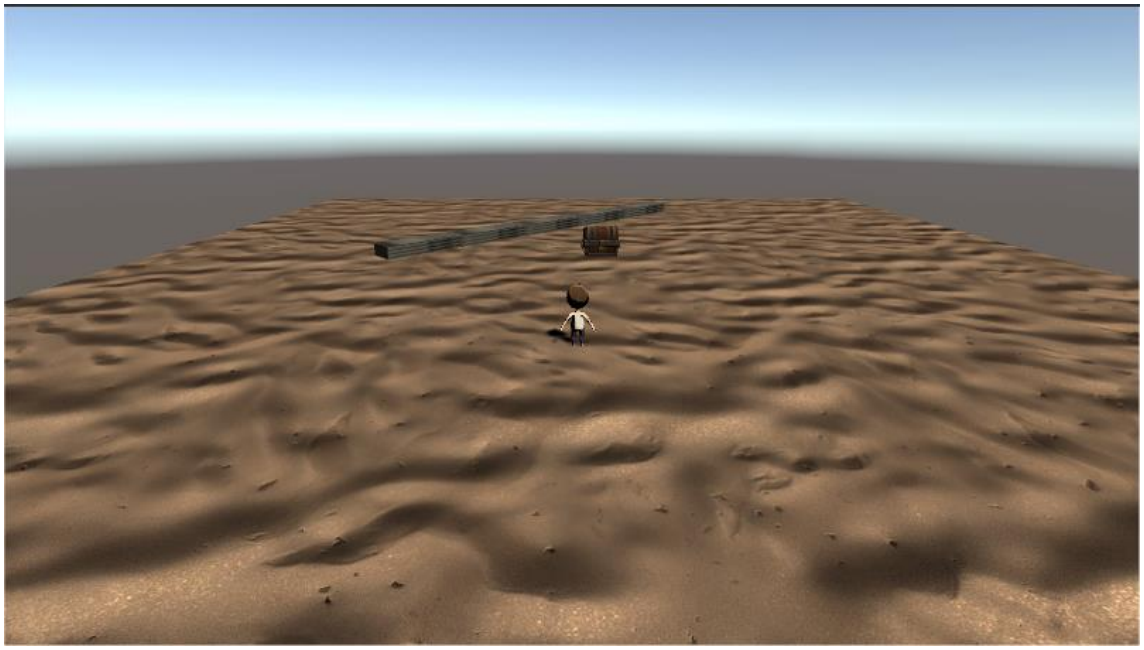

Figura 19: Piso 2 con 2º objetivo (Cofre)

3.2.4 Piso 3

En este caso, se combinan dos elementos, uno nuevo, rampa, y otro visto anteriormente, pared, con el propósito de que el agente entienda que puede sortear una pared subiendo por la rampa, aunque lo mejor sea saltarla. En el diseño del escenario se incluyen cuatro paredes y dos rampas.

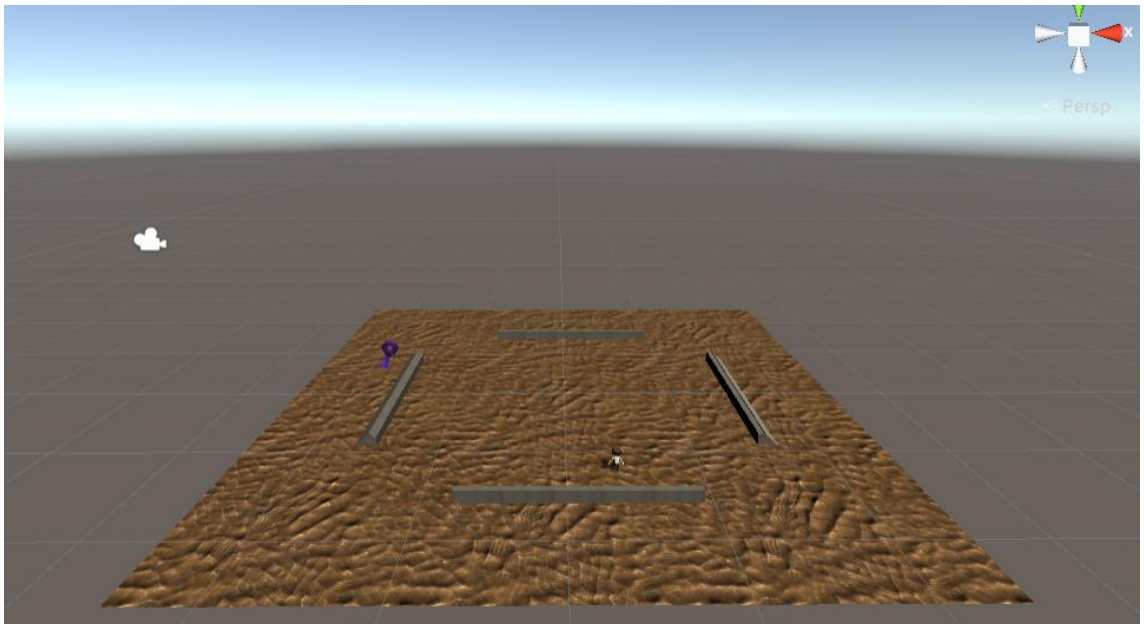

Figura 20: Piso 3 con 1º objetivo (Llave)

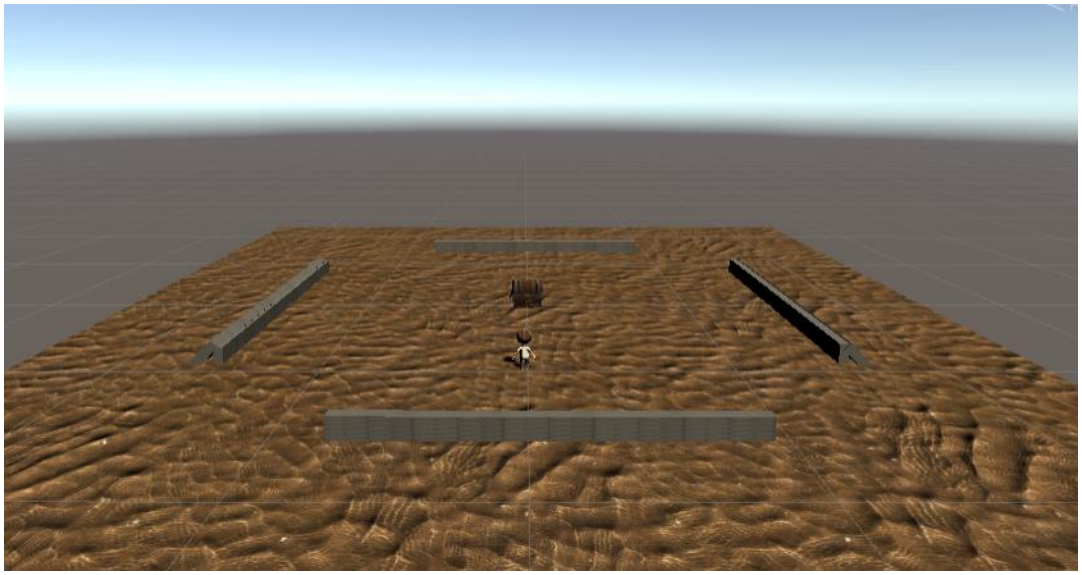

Figura 21: Piso 3 con 2º objetivo (Cofre)

3.2.5 Piso 4

En este piso, la dificultad radica en que se colocan dos muros que el agente, aun saltando, no puede pasar por encima, por lo que su única opción es esquivar los muros. Este escenario se diseñó para evitar que el agente saltase por encima del muro al empezar el piso, ya que inicialmente tiene la opción de saltar habilitada, por lo que cuando se resetea el escenario el agente es capaz de saltar una vez sin necesidad de ver una pared.

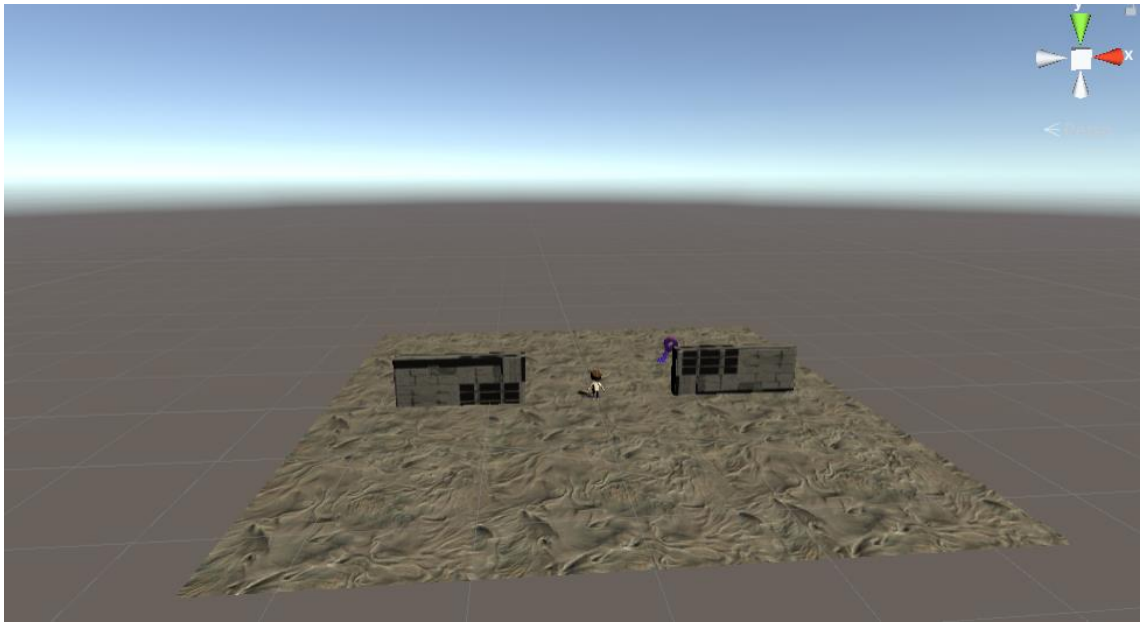

Figura 22: Piso 4 con 1º objetivo (Llave)

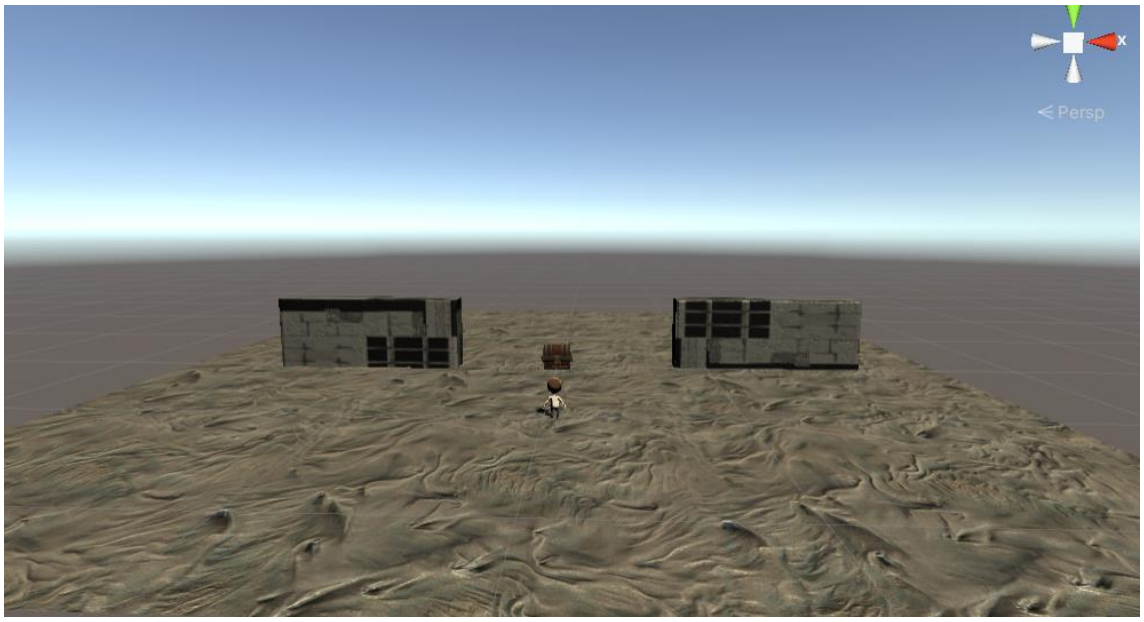

Figura 23: Piso 4 con 2º objetivo (Cofre)

Como ya se ha comentado en el punto anterior, para realizar los entrenamientos es necesario establecer unos parámetros de configuración dependiendo del tipo de algoritmo de aprendizaje que estemos utilizando. En este caso, como primera instancia, se empezó utilizando los parámetros anteriormente mostrados (Figura 9). Como los resultados obtenidos eran inestables se fueron ajustando los valores de los parámetros para obtener mejores resultados. Finalmente se utilizaron los valores de la siguiente figura.

Como se puede observar en estos parámetros de configuración (Figura 24) en comparación con los parámetros iniciales (Figura 8), se han modificado los valores de *learning_rate* y *beta*. La modificación de estos valores permite que el agente aprenda de manera más eficiente. Al disminuir la *beta* se consigue que el agente tome menos decisiones aleatorias lo que hará que los resultados sean más estables, y al disminuir el *learning_rate* cada paso del agente tendrá menos peso en la actualización del descenso de gradiente, lo que se traduce en que el agente aprenderá más lentamente pero su aprendizaje, en este caso, será más fiable.

```
RollerBall:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1000
      buffer_size: 10000
      learning_rate: 9.5e-4
      beta: 3.0e-3
      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: 1000000
    time_horizon: 64
    summary_freq: 10000
```

Figura 24: Parámetros de configuración finales

3.3. Implementación de la lógica del personaje

En este proyecto, para la lógica del personaje, se ha implementado un script con distintos métodos, los cuales explicaremos más adelante, que hacen posible que el agente pueda cumplir su tarea. A través de este script, cada vez que ocurre un evento, actualizamos el escenario, añadimos recompensas y generamos los objetivos.

Para iniciar el videojuego, el script hace uso de dos métodos, *Start()* y *OnEpisodeBegin()*.

En el método *Start()*, que mostraremos a continuación, obtenemos el componente del tipo Rigidbody al que queremos hacer referencia y reseteamos los parámetros. En este caso el objeto Rigidbody se refiere al agente que hemos creado en la escena de Unity.

```
public void Start()
 {
        rBody = GetComponent<Rigidbody>();
        ResetParams = Academy.Instance.EnvironmentParameters;
        SetResetParameters();
 }
```

En el método *OnEpisodeBegin()*, que mostraremos a continuación, inicializamos los valores de las variables a tener en cuenta, además de actualizar la posición de los elementos de cada plataforma, en función de los eventos que se produzcan. Es decir, es donde tenemos en cuenta el avance del personaje, cada vez que coge un objetivo. Para ello se ha implementado un bucle switch que dependiendo del piso en el que nos encontremos y si se ha cogido el primer o segundo objetivo, el agente y los objetivos se generarán en un piso u otro.

```
public override void OnEpisodeBegin()
    {
        maxReward = 0;
        distanciaBuena = 1000f;
        distanciaAnterior = 60f;
        Reward = 0;
        cdjump = 0;
        jumpIsReady = true;
        switch (piso)
        {
            case 0:
                if (this.transform.localPosition.y < 0)
                {
                    this.rBody.angularVelocity = Vector3.zero;
                    this.rBody.velocity = Vector3.zero;
                    this.transform.localPosition      =      new
Vector3(3f, 0.5f, 3f);
                    EpisodeComplete = false;
                    Target2.localPosition = new Vector3(100f, 0f,
100f);
                    Target.localPosition          =          new
Vector3(Random.Range(0f, 24f) * (Random.value <= 0.5 ? 1 : -1),
0.1f, Random.Range(0f, 24f) * (Random.value <= 0.5 ? 1 : -1));
                }
                if (piso == 0)
                {
                    if (EpisodeComplete == false)
                    {
                        Target.localPosition        =        new
Vector3(Random.Range(0f, 24f) * (Random.value <= 0.5 ? 1 : -1),
0.1f, Random.Range(0f, 24f) * (Random.value <= 0.5 ? 1 : -1));
                    }
                    else if (EpisodeComplete == true)
                    {
                        if (segundoObjetivo == 1)
                        {
                            Target2.localPosition      =      new
Vector3(0f, 0f, 0f);
                            Target.localPosition      =      new
Vector3(Random.Range(0f, 24f) * (Random.value <= 0.5 ? 1 : -1),
20.1f, Random.Range(0f, 24f) * (Random.value <= 0.5 ? 1 : -1));
```

```
                    }
                }
            }
            break;
        }
```

Como se puede observar, al inicio del método, se inicializan una serie de variables con el fin de monitorizar el aprendizaje del agente a través de la consola de Unity. Por otro lado, dependiendo del piso en el que nos encontremos, el bucle switch nos llevará a un *case* u otro. En cada *case* se hace lo mismo, al empezar se generan el objetivo y el agente. En el caso de que el agente consiga coger el primer objetivo, se activa el booleano *EpisodeComplete* para que se genere el segundo objetivo. Una vez se consigue el segundo objetivo, se aumenta la variable *piso* para que el bucle switch te lleve al siguiente *case*. En el caso de que el agente se caiga en algún momento durante el transcurso de la tarea, el agente se generará en el centro de la plataforma, de la cual haya caído, y volverá a aparecer el primer objetivo en una posición aleatoria de la plataforma, esto se puede observar en el código, al inicio del *case* donde aparece un bucle *if* cuyo propósito es saber si el agente ha caído de la plataforma.

3.3.1 Sensor de observaciones

Para que el agente aprenda a realizar la tarea correctamente, se le deben dar unas directrices con las que debe guiarse durante el periodo de entrenamiento. Estas directrices son las observaciones que tiene en cuenta el agente. El método utilizado para estas observaciones es *CollectObservations()*, el cual se mostrará a continuación.

```
public override void CollectObservations(VectorSensor sensor)
    {
        if (useVectorObs)
        {
            Vector3               localVelocity                =
transform.InverseTransformDirection(rBody.velocity);
            sensor.AddObservation(Target.localPosition);
            sensor.AddObservation(this.transform.localPosition);
            sensor.AddObservation(localVelocity.x);
            sensor.AddObservation(localVelocity.z);
            sensor.AddObservation(StepCount / MaxStep);
        }
    }
```

En el método *CollectObservations()*, se tienen en cuenta cinco observaciones, la posición del agente, la posición del objetivo, las velocidades del personaje en el eje x y eje z, y por último en qué punto del episodio se encuentra.

Es importante tener en cuenta las velocidades de los ejes porque el agente es un cuerpo rígido por lo que le afectan las físicas, como la inercia, y por lo tanto el agente debe observar el comportamiento del cuerpo al aplicarle una fuerza para poder moverse correctamente. Por otro lado, la observación del punto del episodio en que se encuentra el agente se ha implementado con el propósito de que el agente realice su tarea antes de que se termine el episodio.

Se debe tener presente que cuantas más observaciones le añadamos al método, más complejo va a ser el entrenamiento y por lo tanto puede tardar mucho más en converger a una solución óptima o incluso no converger, por lo que es importante no sobrecargar el método con observaciones innecesarias.

3.3.2 Movimiento del personaje

Para implementar el movimiento del personaje se han utilizado los métodos *OnActionReceived()*, *MoveAgent()* y *Jump()*.

El método *OnActionReceived()*, que se mostrará a continuación, se utiliza para actualizar la recompensa del tiempo que tarda en coger el objetivo y para llamar al método *MoveAgent()*.

```
public override void OnActionReceived(float[] vectorAction)
 {
     AddReward(-1f / MaxStep);
     MoveAgent(vectorAction);
 }
```

Como se puede observar, este método, añade una recompensa negativa en cada paso del videojuego y seguidamente llama al método *MoveAgent()* al que le pasa el vector de acciones que genera.

En el método *MoveAgent()*, que se mostrará a continuación, es donde se toman las decisiones de qué acciones realizar en función del vector que se ha recibido, además de otras funciones que se expondrán más adelante. En nuestro caso el personaje puede realizar cuatro acciones hacia delante o hacia atrás, izquierda o derecha, rotar izquierda o rotar derecha y saltar.

```
public void MoveAgent(float[] act)
    {
        float                distanceToTarget2                =
Vector3.Distance(this.transform.localPosition,
Target2.localPosition);
        float                 distanceToTarget                =
Vector3.Distance(this.transform.localPosition,
Target.localPosition);
        var dirToGo = Vector3.zero;
        var rotateDir = Vector3.zero;
        if (Mathf.FloorToInt(act[0]) == -1)
        {
            dirToGo = transform.forward;
        }
        else if (Mathf.FloorToInt(act[0]) == 1)
        {
            dirToGo = -transform.forward;
        }
        if (Mathf.FloorToInt(act[1]) == -1)
        {
            dirToGo = transform.right;
        }
        else if (Mathf.FloorToInt(act[1]) == 1)
        {
            dirToGo = -transform.right;
        }
        if (Mathf.FloorToInt(act[2]) == -1)
        {
            rotateDir = transform.up;
        }
        else if (Mathf.FloorToInt(act[2]) == 1)
        {
```

```
            rotateDir = -transform.up;
        }
        if (Mathf.FloorToInt(act[3]) == 1)
        {
            if (Time.time > cdjump + 2f)
            {
                Jump();
            }
        }
        rBody.AddForce(dirToGo * forceMultiplier,
ForceMode.VelocityChange);

rBody.MoveRotation(Quaternion.Euler(rBody.rotation.eulerAngles +
rotateDir * turnSpeed));

        if (Physics.Raycast(this.transform.position,
Vector3.forward, out hit, 5f))
        {

            if (hit.transform.CompareTag("Pared"))
            {
                jumpIsReady = true;
            }
        }
```

Como se puede observar en el código, este método implementa una serie de bucles if-else anidados en los que, dependiendo de los valores que recibe como argumento, se actualizarán los vectores de movimiento que se crean al principio del método, *dirToGo* y *rotateDir*¸ para posteriormente utilizarlos para añadir fuerzas a nuestro agente y que este pueda realizar los movimientos oportunos. Además, se puede observar como al principio del método se calculan las distancias que hay desde el agente al objetivo, estas sentencias se explicaran con más detalle en el siguiente apartado.

El método *Jump()* solo se utilizar para saltar, siempre y cuando el salto este disponible, ya que no se quiere que pueda saltar siempre sino solo cuando vea una pared a una cierta distancia. Además se implementa un tiempo de reutilización para volver a saltar.

```
  private void Jump()
    {
        if (jumpIsReady)
        {
            rBody.AddForce(new Vector3(0f, jumpForce + 2f, 0f),
ForceMode.VelocityChange);
            cdjump = Time.time;
            jumpIsReady = false;
        }
    }
```

Como se puede observar en el código, cada vez que el agente llame al método *Jump()*, primero se comprobará si la variable booleana *jumpIsReady* esta activada para que el agente pueda saltar. Esta variable estará activada por defecto al iniciar el videojuego, pero una vez salte, el agente, se desactivará y solo se volverá a activar cuando el agente vea una pared a una cierta distancia. De esta forma se evita que el agente esté constantemente saltando. Por otro lado, una vez salte, se actualizará el tiempo de reutilización para que no pueda saltar dos veces seguidas hasta que no

hayan pasado al menos dos pasos desde el último salto, de esta forma se evita que el agente pueda volver a saltar durante el salto.

3.3.3 Recompensas

Las recompensas son un punto importante a la hora de entrenar cualquier agente. Un mal sistema de recompensas puede llevar a resultados erráticos. Por otro lado, no es complicado establecer un sistema de recompensas viable.

En nuestro caso las recompensas se han divido en tres métodos *OnActionReceived()*, *MoveAgent()* y *OnCollisionEnter()*.

Como ya se ha adelantado en el apartado 4.3.1, en el método *OnActionReceived()*, se le va sumando una recompensa negativa, cada vez mayor, en función de los pasos que da el agente antes de completar la tarea. Esta recompensa se añade con el fin de que el personaje resuelva el escenario lo antes posible.

```
AddReward(-1f / MaxStep);
```

Como se comentó en el apartado anterior, al inicio del método *MoveAgent()*, se calculan las distancias del agente a cada uno de los objetivos. Estas distancias son las utilizadas para saber si el agente ha conseguido coger cualquier objetivo. Por otra parte, estas distancias también se utilizan para saber si el agente se ha movido o permanece quieto. En caso de que el agente permanezca quieto se le añade una recompensa negativa muy pequeña para que aprenda que debe moverse constantemente.

```
if (distanceToTarget != distanciaAnterior)//intento evitar que se
quede quieto en un sitio
        {
            if (distanceToTarget < 1.3)
            {
                if (objetivo == 10)
                {
                    SetReward(10f);
                    Reward += 10;
                    EpisodeComplete = true;
                    segundoObjetivo++;
                    EndEpisode();
                    objetivo = 0;
                    objetivoAcumulado++;
                }
                else if (objetivo == 5)
                {
                    SetReward(3f);
                    Reward += 3;
                    EpisodeComplete = true;
                    segundoObjetivo++;
                    EndEpisode();
                    objetivo++;
                    objetivoAcumulado++;
                }
                else
                {
                    SetReward(1f);
                    Reward += 1;
```

```
                EpisodeComplete = true;
                segundoObjetivo++;
                EndEpisode();
                objetivo++;
                objetivoAcumulado++;
            }
        }
        distanciaAnterior = distanceToTarget;
    }
    else
    {
        AddReward(-1f);
        Reward -= 1f;
    }
    if (distanceToTarget2 != distanciaAnterior2)
    {
        if (distanceToTarget2 < 1.3)
        {
            SetReward(2f);
            Reward += 2;
            EpisodeComplete = true;
            segundoObjetivo++;
            if (piso < 4)
            {
                piso++;
                segundoObjetivo = 0;
                EndEpisode();
            }
            else if (piso == 4 && objetivo == 4)
            {
                SetReward(100f);
                Reward += 100f;
                segundoObjetivo = 2;
                EndEpisode();
            }
            else if (piso == 4)
            {
                SetReward(10f);
                Reward += 10f;
                segundoObjetivo = 2;
                EndEpisode();
            }
        }
        distanciaAnterior2 = distanceToTarget2;
    }
    else
    {
        AddReward(-1f);
        Reward -= 1f;
    }
```

Como se puede observar en este código, las recompensas se han implementado para que cuando el agente se acerque más de 1,3 metros, a cualquier objetivo, reciba una recompensa de +1 en el caso de coger la llave y +2 en el caso de coger el cofre. De esta forma el agente aprenderá a acercarse al objetivo cada vez que lo detecte. Además, para que el agente aprenda a coger los

objetivos seguidos, se le proporciona una recompensa positiva de +5 cuando consigue coger 5 veces seguidas el primer objetivo y se le añade otra recompensa positiva de +10 si el agente es capaz de coger diez veces seguidas el primer objetivo sin caerse. También, para favorecer que el agente termine la torre de pisos, se le añade una recompensa por completar la torre de +10. Además, para que complete todos los pisos sin caer se le añade una recompensa de +100 por resolverlo perfecto.

Todas estas recompensas extra se le añaden con el fin de que reducir el tiempo de entrenamiento del agente. Con los parámetros de configuración adecuados y el tiempo suficiente de entrenamiento, el agente debería aprender esta lógica, por sí mismo, sin problemas.

```
  if (this.transform.localPosition.y < 0 && piso == 0)
        {
            SetReward(-10.0f);
            Reward -= 10.0f;
            segundoObjetivo = 0;
            piso = 0;
            fall++;
            objetivo = 0;
            segundoObjetivo = 0;
            EpisodeComplete = false;
            EndEpisode();
        }
        else if (this.transform.localPosition.y < 20 && piso ==
1)
        {
            SetReward(-10.0f);
            Debug.Log("Cae de la plataforma 1: -10");
            Reward -= 10.0f;
            segundoObjetivo = 0;
            piso = 0;
            fall++;
            objetivo = 0;
            segundoObjetivo = 0;
            EpisodeComplete = false;
            EndEpisode();
        }
        else if (this.transform.localPosition.y < 40 && piso ==
2)
        {
            SetReward(-10.0f);
            Debug.Log("Cae de la plataforma 2: -10");
            Reward -= 10.0f;
            segundoObjetivo = 0;
            piso = 0;
            fall++;
            objetivo = 0;
            segundoObjetivo = 0;
            EpisodeComplete = false;
            EndEpisode();
        }
        else if (this.transform.localPosition.y < 60 && piso ==
3)
        {
            SetReward(-10.0f);
            Debug.Log("Cae de la plataforma 3: -10");
```

```
            Reward -= 10.0f;
            segundoObjetivo = 0;
            piso = 0;
            fall++;
            objetivo = 0;
            segundoObjetivo = 0;
            EpisodeComplete = false;
            EndEpisode();
        }
        else if (this.transform.localPosition.y < 80 && piso ==
4)
        {
            SetReward(-10.0f);
            Debug.Log("Cae de la plataforma 4: -10");
            Reward -= 10.0f;
            segundoObjetivo = 0;
            piso = 0;
            fall++;
            objetivo = 0;
            segundoObjetivo = 0;
            EpisodeComplete = false;
            EndEpisode();
        }

    }
```

Como se puede apreciar en este fragmento de código, para que el agente aprenda a permanecer en la plataforma se le añade una recompensa de -10 cada vez que se caiga de una plataforma, de esta manera cuando el agente caiga de la plataforma y reciba una recompensa muy grande negativa, el agente, poco a poco, se dará cuenta de que debe permanecer dentro de la plataforma.

En el método *OnCollisionEnter()*, se establecen recompensas negativas al colisionar con los elementos del videojuego tales como una pared, un muro y una rampa, con el fin de que el agente aprenda a evitar tanto el muro como la rampa y que salte la pared. Cada vez que colisiona contra una pared se le añade una recompensa de -0.025, si es una rampa de -0.015 y si es un muro de -0.5. De esta forma el agente aprenderá a no chocar con los elementos del escenario y los bordeará.

```
private void OnCollisionEnter(Collision collidedObj)
    {
        if (collidedObj.gameObject.CompareTag("Pared"))
        {
            jumpIsReady = true;
            AddReward(-0.025f);
            Reward -= 0.025f;
        }
        if (collidedObj.gameObject.CompareTag("Muro"))
        {
            AddReward(-0.5f);
            Reward -= 0.5f;
        }
        if (collidedObj.gameObject.CompareTag("Rampa"))
        {
            AddReward(-0.015f);
            Reward += -0.015f;
        }
    }
```

En este método se puede observar que para que el agente diferencie el objeto contra el que colisiona es necesario, previamente, haber establecido un Tag, en el entorno de Unity, para cada elemento del escenario. Por otra parte, se puede ver como cuando el agente choca contra una pared se le habilita el booleano *jumpIsReady* para que el agente pueda saltar.

4. Pruebas realizadas

En este capítulo se expondrán las pruebas realizadas para ajustar los parámetros de configuración. Además, se explicará cómo se ha divido el entrenamiento de dos modos distintos. Uno se basa en el entrenamiento del entorno con un único piso en cada entramiento. Mientras que el otro se basa en el entrenamiento del entorno añadiendo pisos cada vez que se aprenda el anterior, de manera que se empezará entrenando con un solo piso y se acabará entrenando a los cinco pisos juntos. Para monitorizar los entrenamientos se hará uso de la herramienta TensorBoard, la cual nos proporciona unas gráficas de los resultados durante el entrenamiento. Estas gráficas nos darán una visión de cómo de exitosos han sido los entrenamientos. En los siguientes apartados, se mostrarán cuatro gráficas en concreto, en todas ellas el eje X reprensentará los pasos que lleva el entrenamiento. Estas gráficas muestran métricas para medir el rendimiento del entrenamiento. Para que unos resultados sean exitosos la gráfica de recompensa acumulada debe tener una tendencia ascendente, la del tiempo medio en resolver el entorno debe aumentar al principio y terminar con tendencia descendente o constante, la gráfica de la beta, que controla si las políticas que se van eligiendo son más o menos aleatorias en cuanto a la elección de acciones, debe disminuir y por último la gráfica de la entropía debe aumentar al inicio y terminar en tendencia descendente al acabar el entrenamiento.

4.1 Entrenamiento para ajustar parámetros

Como ya se comentó en capítulos anteriores, es necesario ajustar los parámetros de configuración para que el entrenamiento del agente sea eficiente. En este proyecto, como se utiliza el algoritmo PPO, los parámetros de configuración a tener en cuenta son *Beta* y *Epsilon*. En este caso, también se va a variar el parámetro *learning_rate*, común para todos los algoritmos de aprendizaje. Por lo tanto, mediante la modificación de estos valores podremos ajustar el entrenamiento para que sea lo mejor posible.

BETA	Nº DE VECES QUE COGE EL OBJETIVO	Nº DE VECES QUE CAE	RELACIÓN OBJ/CAE
$1.0E^{-4}$	4833	1606	3.009
$2.0E^{-3}$	4722	1405	3.36
$3.0E^{-3}$	4274	1165	3.668
$4.0E^{-3}$	4867	1345	3.618
$9.0E^{-3}$	4793	1327	3.611
$8.0E^{-3}$	4593	1339	3.430
$7.0E^{-3}$	4676	1536	3.044
$6.0E^{-3}$	5003	1491	3.355
$5.0E^{-3}$	6943	2265	3.065
$9.6E^{-3}$	4557	1293	3.524
$9.5E^{-3}$	5233	1451	3.606

9.3E^{-3}	5124	1402	3.654
9.1E^{-3}	4709	1334	3.529
9.4E^{-3}	4437	1310	3.387

Tabla 2: Valores de Beta entrenados

Como se puede observar en esta tabla (tabla 2), el valor de Beta que mejores resultados ha conseguido ha sido Beta= 9.3e-3. Se partirá de esta Beta para realizar los entrenamientos sucesivos.

LEARNING_RATE	Nº DE VECES QUE COGE EL OBJETIVO	Nº DE VECES QUE CAE	RELACIÓN OBJ/CAE
4.0E^{-4}	1006	2240	0.449
9.0E^{-4}	1243	2288	0.543
1.0E^{-5}	694	12798	0.054
9.5E^{-4}	1482	1812	0.817
9.6E^{-4}	1591	2010	0.791
9.7E^{-4}	1081	2103	0.514
9.65E^{-4}	933	1833	0.509
9.55E^{-4}	1076	2193	0.490
9.62E^{-4}	1482	1984	0.746
5.9E^{-4}	1493	2025	0.737
6.0E^{-4}	1702	2147	0.792

Tabla 3: Valores de Learning_rate entrenados

Como se puede observar en esta tabla (tabla 3), el valor de learning_rate que mejores resultados ha obtenido ha sido 6.0e-4, por lo tanto, se partirá de este valor para entrenar los diferentes pisos.

En cuanto a Epsilon, se entrenó con tres valores distintos, 0.1, 0.2 y 0.3. Se concluyó que el valor óptimo de entrenamiento para este tipo de videojuego era Epsilon = 0.2, tanto en rendimiento como en duración del entrenamiento. Para ambos casos los mejores resultados se obtenían con Epsilon= 0.2.

4.2 Entrenamiento de pisos por separados

En un principio se decidió entrenar cada piso por separado, de manera que el agente aprendiera a resolver un piso, o nivel, y partiendo de esa base, pudiera aprender el siguiente. De modo que se crea una red neuronal del entrenamiento del piso 0 y a partir de esa red se iban entrenando los siguientes pisos, es decir, primero se entrena al agente en el piso 0, después se utilizará esa red, que ya ha aprendido a resolver el piso 0, en el piso1, a continuación se usará esta red neuronal, que ha aprendido a resolver el piso 1 partiendo del piso 0, como punto de partida del piso 2 y así sucesivamente. Por otro lado, en este primer método de entrenamiento se utilizará un sistema de

recompensas diferente al del siguiente apartado. En este caso las recompensas serán de diez veces superiores a las del apartado siguiente, es decir, en vez de sumar un punto cada vez que se obtiene el objetivo se sumarán diez puntos y en vez de restar diez puntos cada vez que se cae de la plataforma se restarán cien. Debido a esto, en un principio se verá, en las gráficas, que la recompensa supera el valor cero de forma habitual, pero en el siguiente apartado será raro que la recompensa supere el valor cero. En cualquier caso, se realizaron pruebas con los dos sistemas de recompensas y se comprobó que mientras se mantuviesen las recompensas en el mismo orden el resultado de los entrenamientos no varía.

4.2.1 Piso 0

En este primer piso lo resultados obtenidos del entrenamiento servirán como referencia de los entrenamientos de los pisos sucesivos.

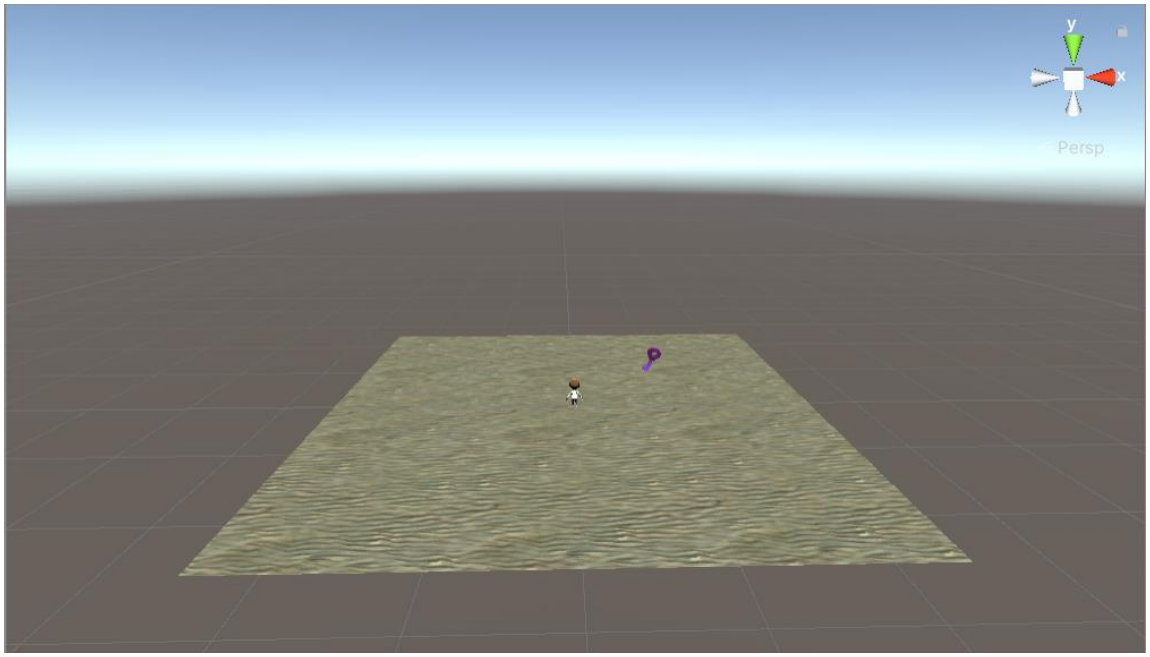

Figura 25: Piso0

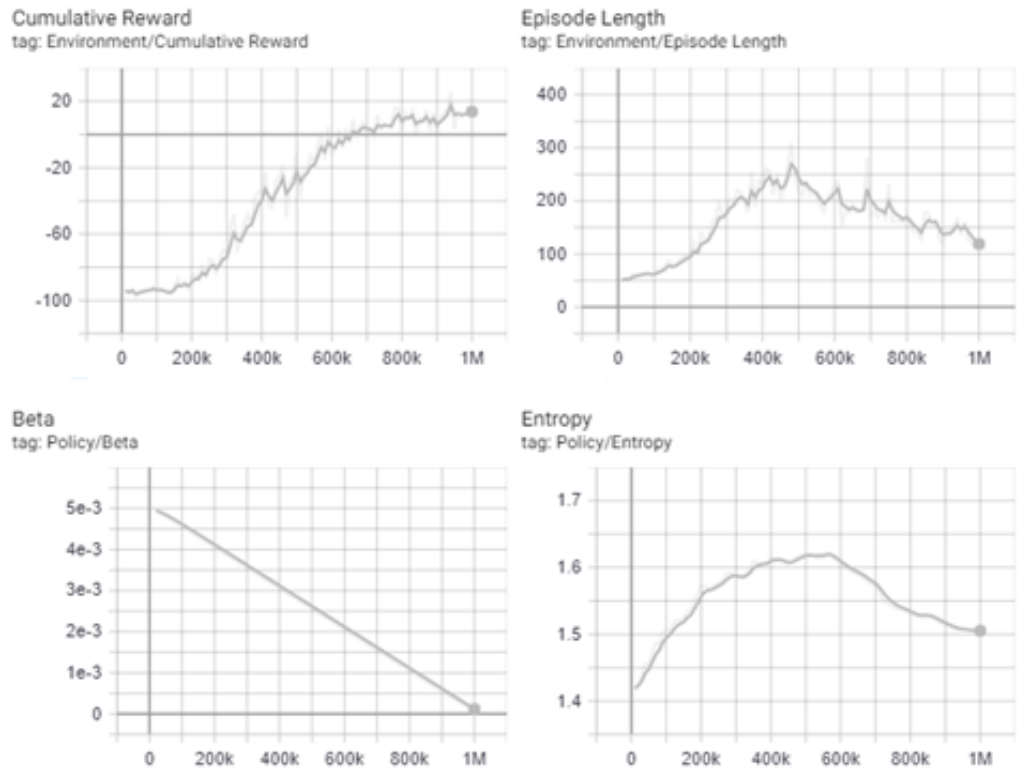

Figura 26: Resultados piso0

Como se puede observar en la (Figura 26), los resultados obtenidos son buenos. La recompensa acumulada tiene una tendencia creciente, el tiempo medio en terminar los episodios al principio aumenta hasta alcanzar un óptimo y después va disminuyendo y la entropía realiza una curva casi óptima.

4.2.2 Piso1

Partiendo de la red neuronal anterior, se entrena al agente en este escenario (Figura 26).

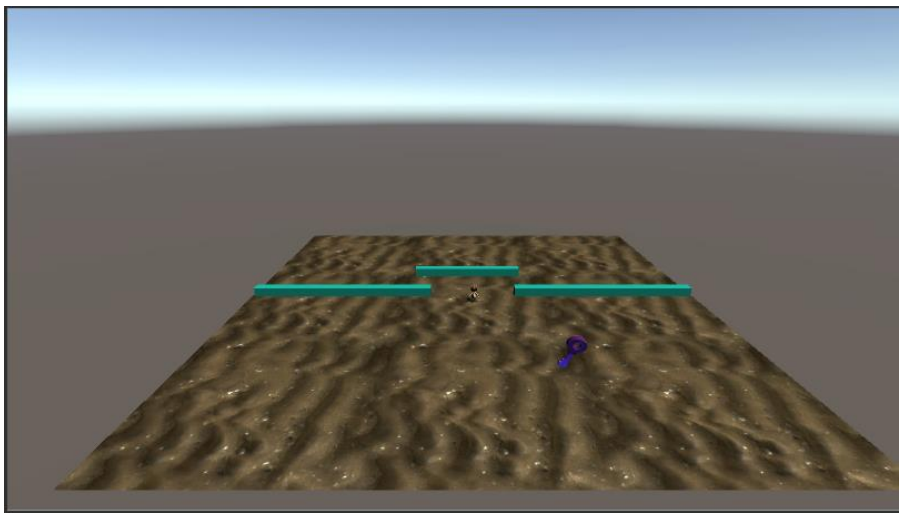

Figura 27: Piso1

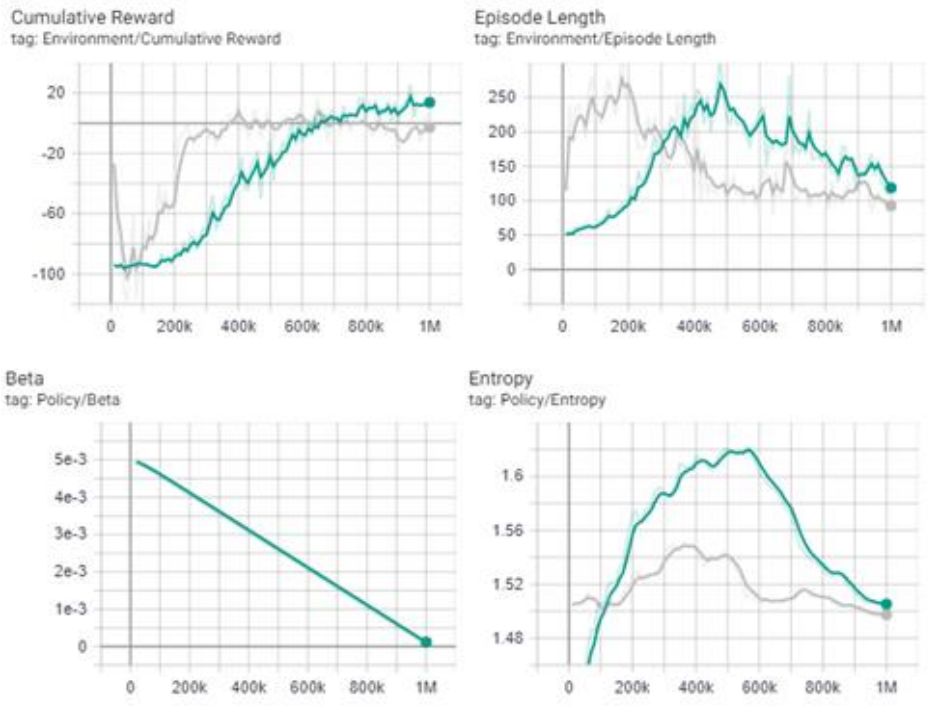

Figura 28: Resultados piso0(en verde) vs piso1(en gris)

Como se puede observar en esta imagen (figura 28), los resultados de las gráficas son peores que los anteriores, pero siguen siendo correctas. Este cambio en los resultados se debe a la dificultad incrementada del piso 1. Se puede deducir que el agente emplea más tiempo inicialmente ya que ha aprendido a no caerse, es por eso que la gráfica del tiempo medio en terminar un episodio tarda más en empezar a descender.

4.2.3 Piso2

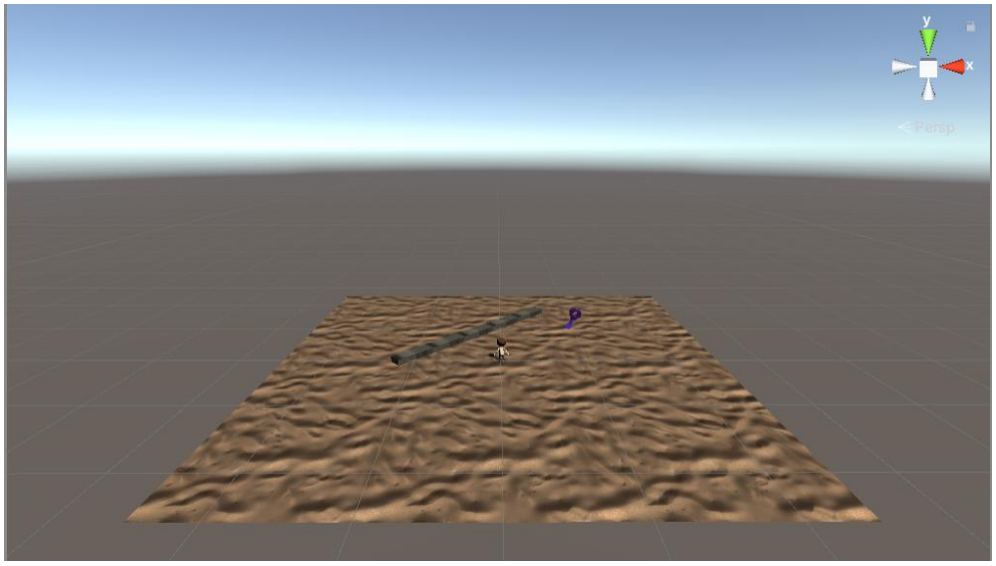

Figura 29: Piso2

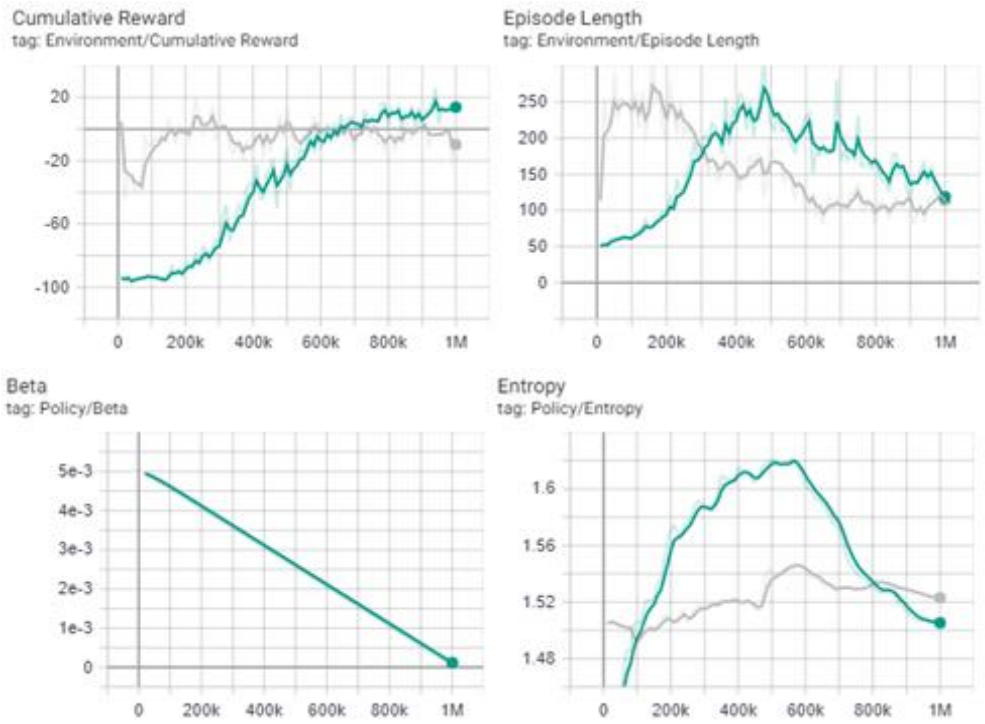

Figura 30: Resultados piso0(en verde) vs piso2(en gris)

Como se pueden observar en estos resultados (Figura 30), parece que al agente le cuesta mejorar su aprendizaje, aunque se mantiene constante en la realización de la tarea, es decir, completa su misión, pero va empeorando el rendimiento.

4.2.4 Piso3

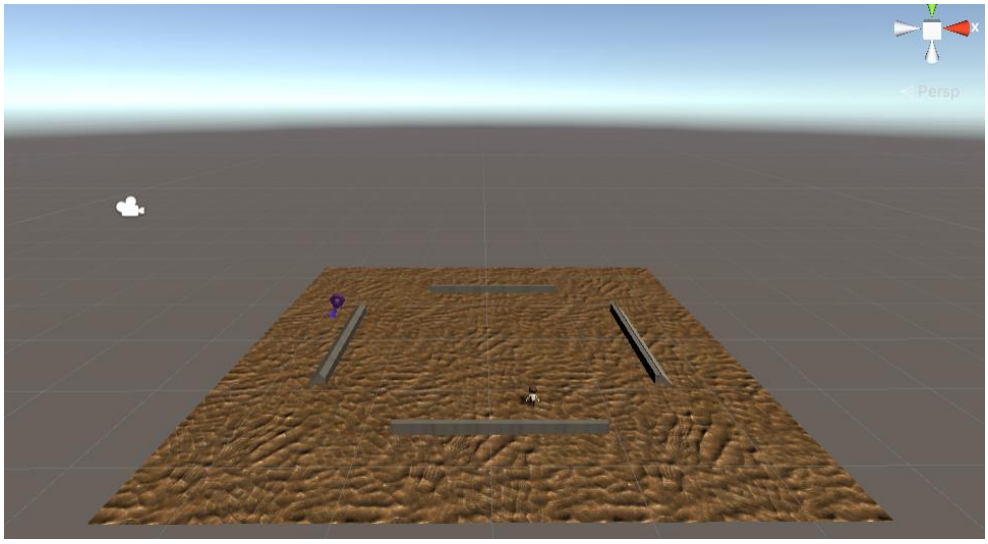

Figura 31: Piso3

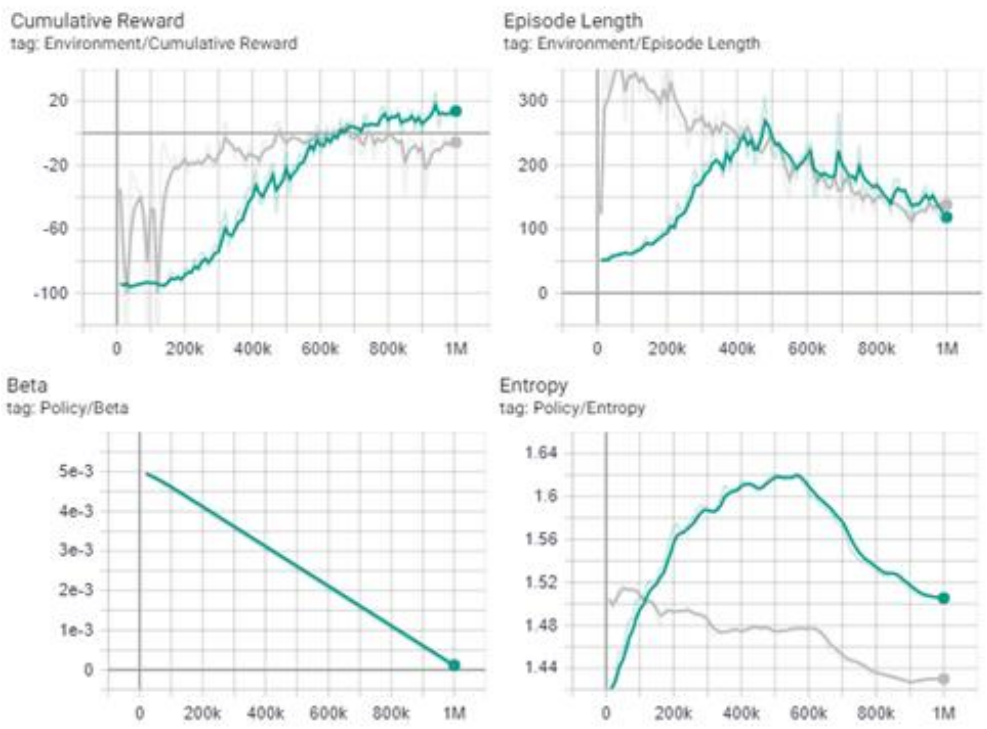

Figura 32: Resultados piso0(en verde) vs piso3(en gris)

Como se puede ver en esta imagen (Figura 32), los resultados son ligeramente mejores que en el anterior piso. Parece que el agente tiene menos dificultades para superar este nivel que el anterior. Esto se debe a que en este nivel el agente puede saltar, lo que le permite ver el objetivo detrás de las paredes y así dirigirse a él.

4.2.5 Piso4

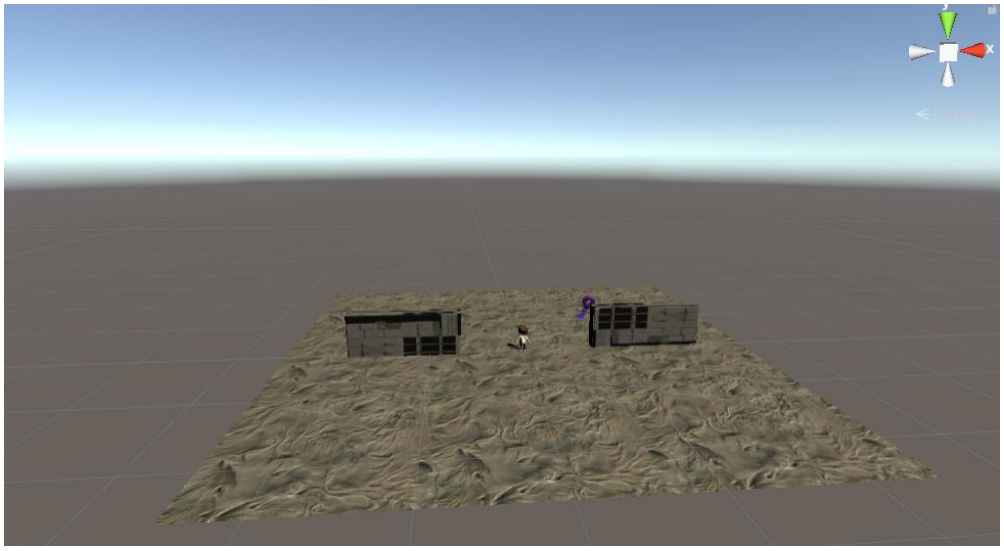

Figura 33: Piso4

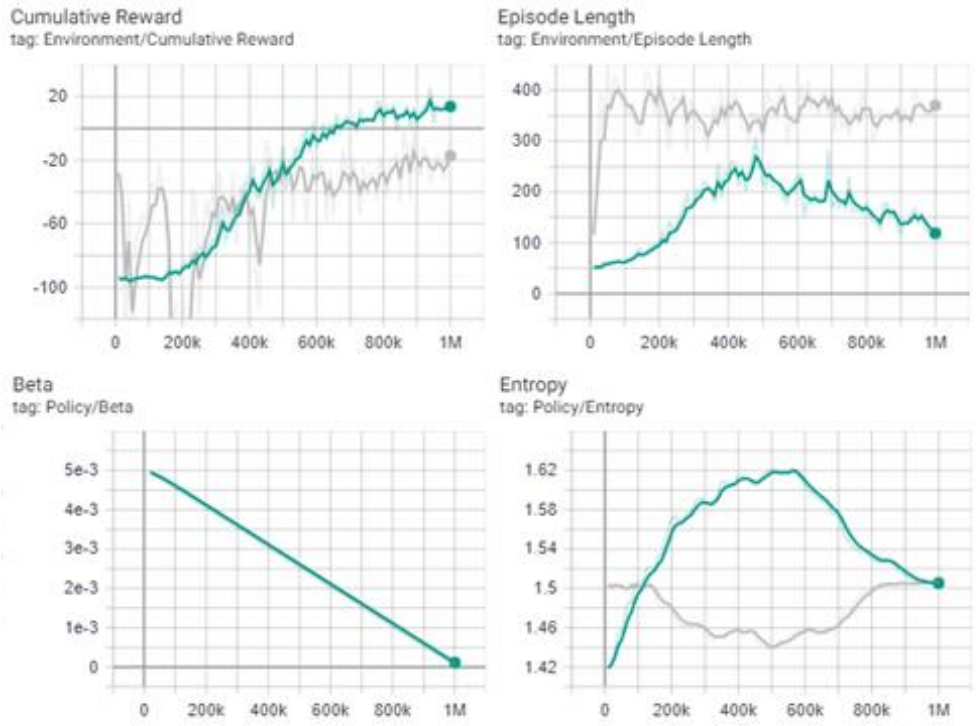

Figura 34: Resultados piso0(en verde) vs piso4(en gris)

Como se puede observar (figura 34), el agente tiene problemas en resolver este nivel, ya que el tiempo medio en terminar el episodio es muy alto y la entropía realiza una curva inversa a la que debería realizar en términos ideales.

Basándose en estos resultados, se puede apreciar como el agente disminuye su rendimiento a lo largo del desarrollo del proyecto. Esto propició que se entrenara al agente de otra forma.

4.3 Entrenamiento añadiendo pisos en un mismo entorno

Como se puede observar en las imágenes del apartado anterior, al principio el entrenamiento tiene buenos resultados, pero al incrementarle la dificultad poco a poco van empeorando. Esto es algo normal ya que los pisos cada vez son más difíciles. Lo preocupante es que la entropía de los entrenamientos solo va en aumento y debería subir y bajar por lo que al final se decidió cambiar los parámetros de configuración y se probó a entrenarlo de otra forma. Esta vez, se optó por ir añadiendo los pisos conforme se entrenaba cada uno, es decir, el primer entrenamiento tendrá solo el piso 0, el siguiente entrenamiento tendrá el piso 0 y el piso 1 en la misma escena, después del piso 0 al piso 2 y así hasta terminar los cinco pisos que conforman la torre. Cabe destacar que cada vez que el agente cae de la plataforma vuelve a aparecer en el piso 0, de esta forma a medida que avance el entrenamiento la dificultad aumentará en mayor medida.

4.3.1 piso0

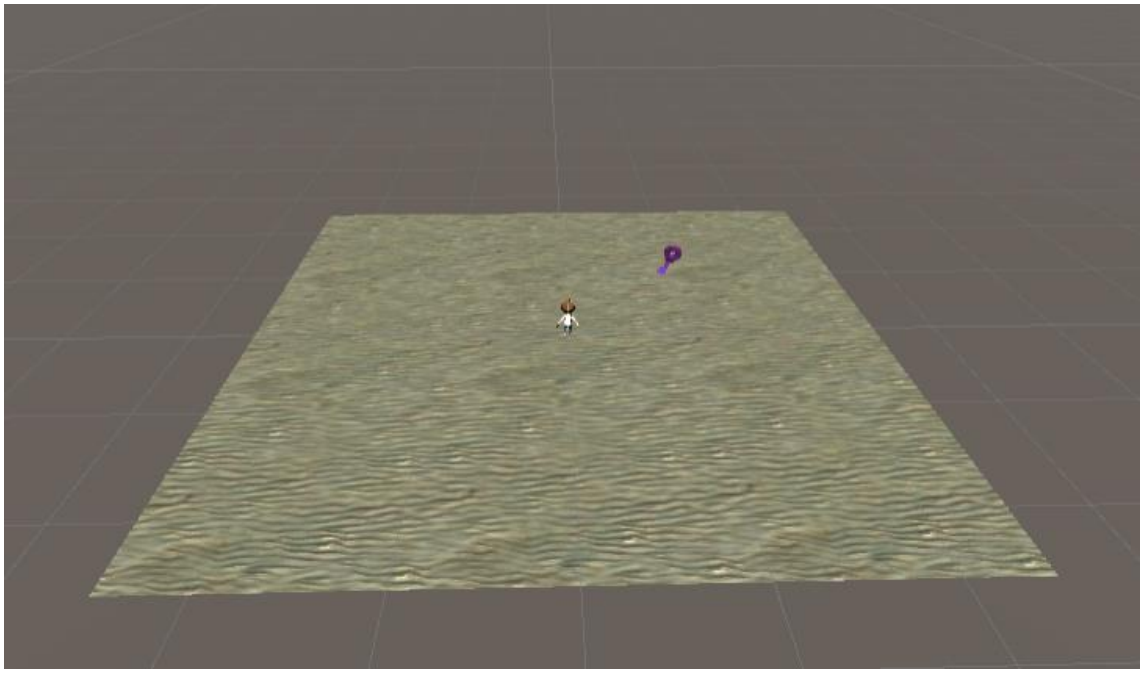

Figura 35: Piso0

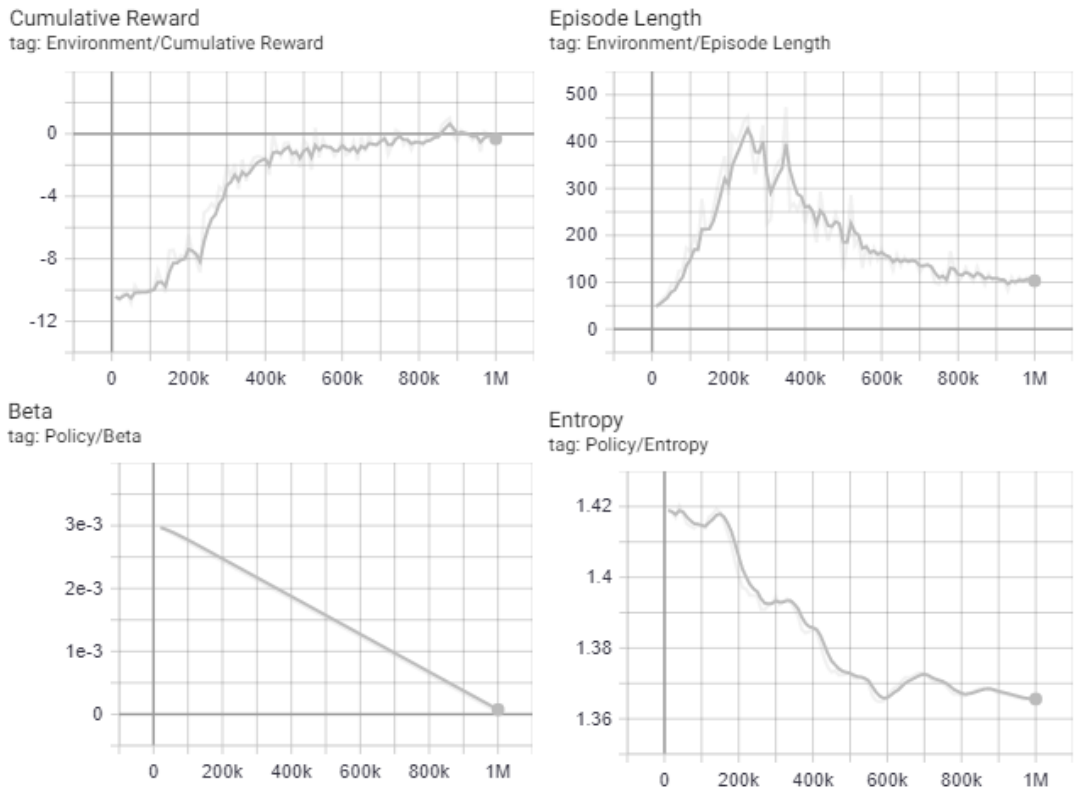

Figura 36: Resultados piso 0 de Tensorboard.

Como se puede observar en la imagen (figura 36), la recompensa acumulada va incrementándose a lo largo del entrenamiento, la entropía disminuye, como debe ser para que el entrenamiento sea

exitoso, al igual que el tiempo en terminar los episodios. Podemos afirmar que los resultados de este entrenamiento son correctos.

4.3.2 Del Piso0 al Piso1

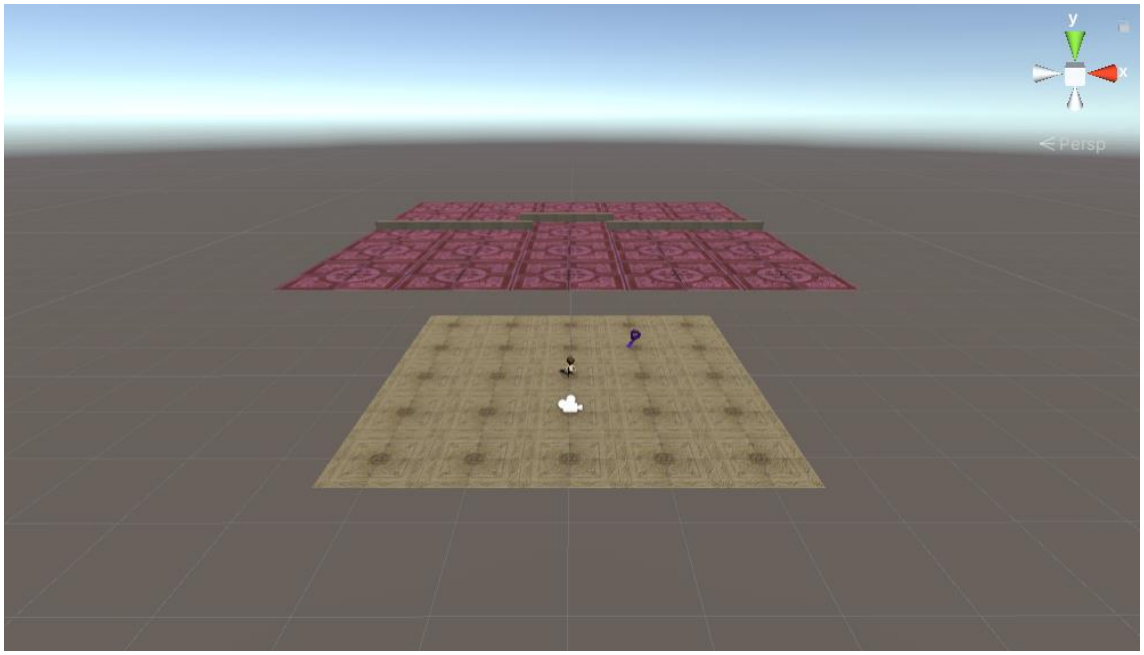

Figura 37: Del piso0 al piso1

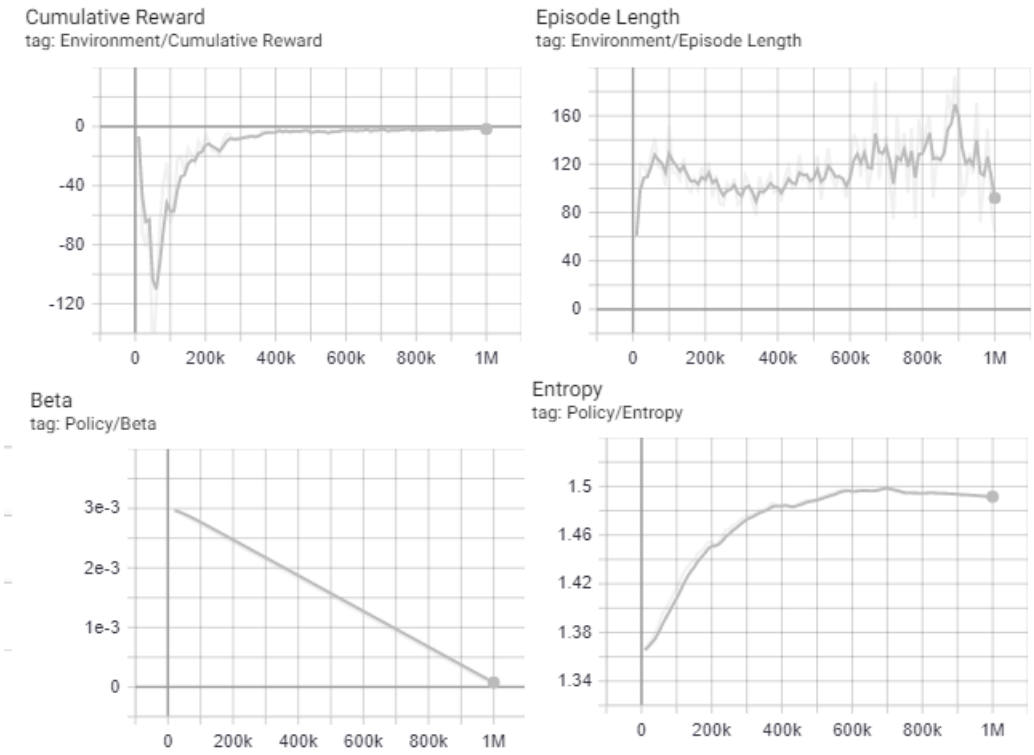

Figura 18: Resultados Tensorboard del piso0-piso1

Como se puede apreciar en la figura 38, el resultado del entrenamiento es bueno, aunque un poco peor que el de la figura 30. En este caso se puede ver que el agente ya tiene aprendido el piso 0 y

es por esto que el tiempo medio en terminar los episodios es bajo. Por otro lado, se puede observar, en la gráfica de la entropía, como al principio del entrenamiento el agente se encuentra con un nuevo entorno por lo que la curva es creciente, y al final de la misma se ve como empieza a bajar.

Estos resultados ponen de manifiesto que el entrenamiento ha sido bueno en general, aunque podría haber sido mejor.

4.3.3 Del Piso0 al Piso2

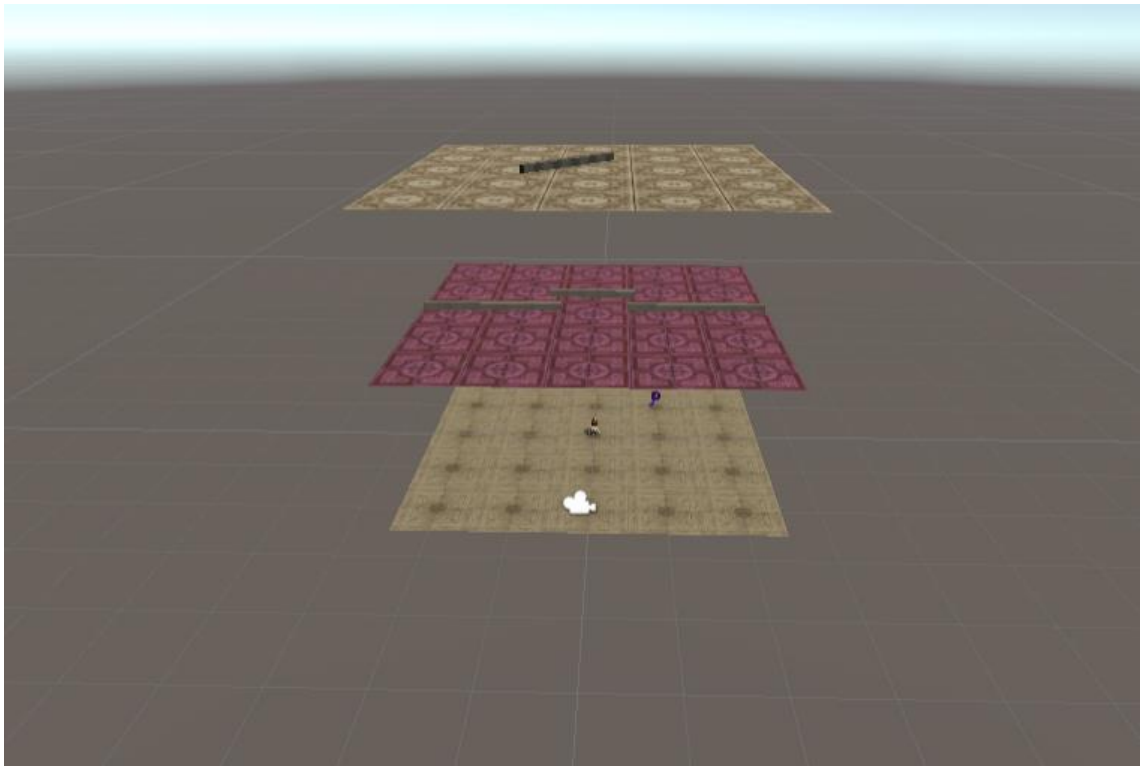

Figura 39: Del piso0 al piso2

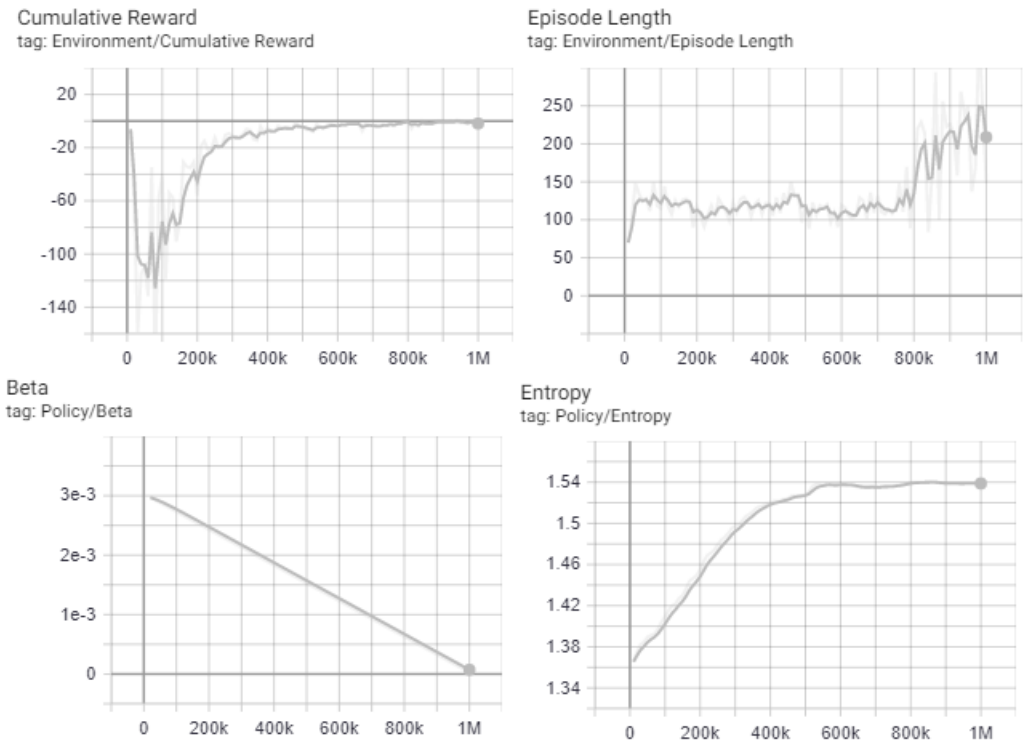

Figura 40: Resultados de Tensorboard del piso 0 al piso 2

Como se puede observar en esta imagen (figura 40), el resultado del entrenamiento es bueno en general, pero la gráfica de la entropía debería bajar. Esto no quiere decir que el entrenamiento no haya sido exitoso, pero da a entender que existen unos valores de configuración más adecuados para este tipo de entorno.

4.3.4 Del Piso0 al Piso3

A partir de este entrenamiento los resultados fueron parecidos hasta llegar el piso3 el cual muestro a continuación.

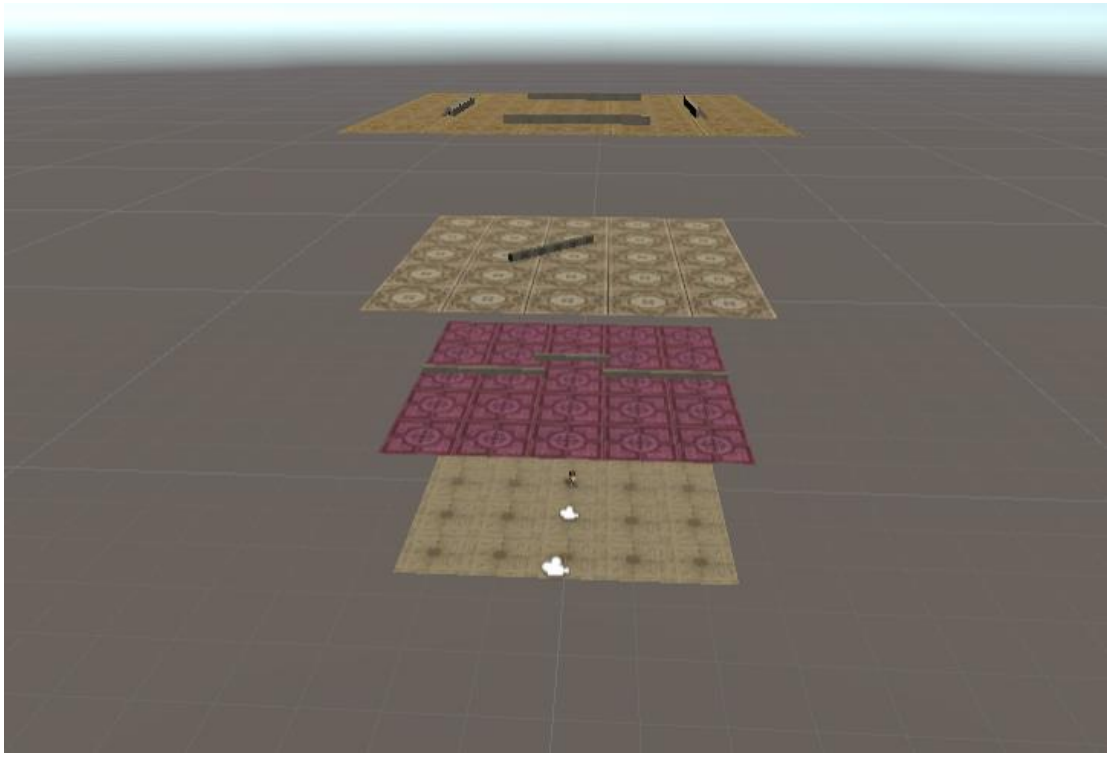

Figura 41: Del piso0 al piso3

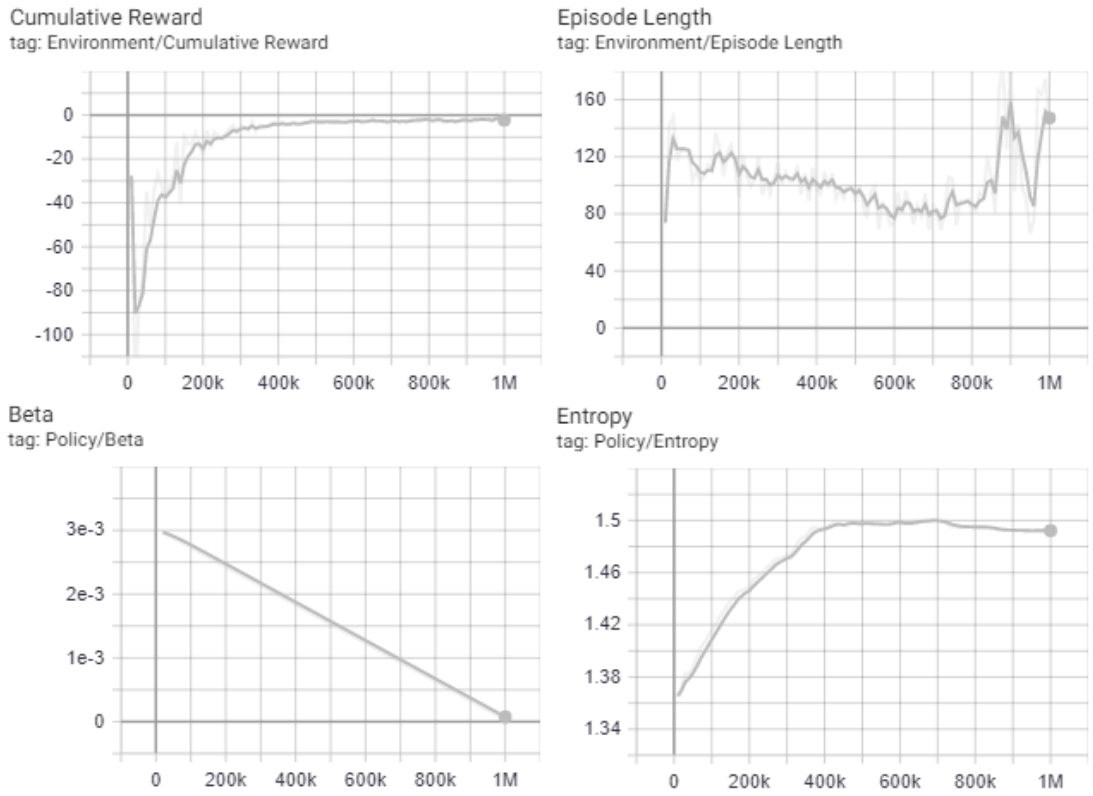

Figura 42: Piso0-piso3 en tensorboard.

En esta imagen (figura 42) se puede observar que el entrenamiento se ha desarrollado correctamente. En cuanto al tiempo medio por episodio, se puede ver que al final del entrenamiento el tiempo aumenta, esto es debido a que el agente se queda atrapado en el último piso y le cuesta más encontrar los objetivos. Aun así, el tiempo medio en terminar los episodios

es relativamente bajo. En cuanto a la entropía, al igual que el caso anterior, se puede ver que no llega a bajar, lo que pone de manifiesto que el agente sigue probando distintas políticas para ajustar su aprendizaje para acercarse a la solución óptima. Los resultados podrían ser mejores, pero se encuentran dentro de los límites para ser aceptados, además de que el agente consigue terminar los cuatro pisos sin caerse más de una vez.

4.3.5 Del Piso0 al Piso4

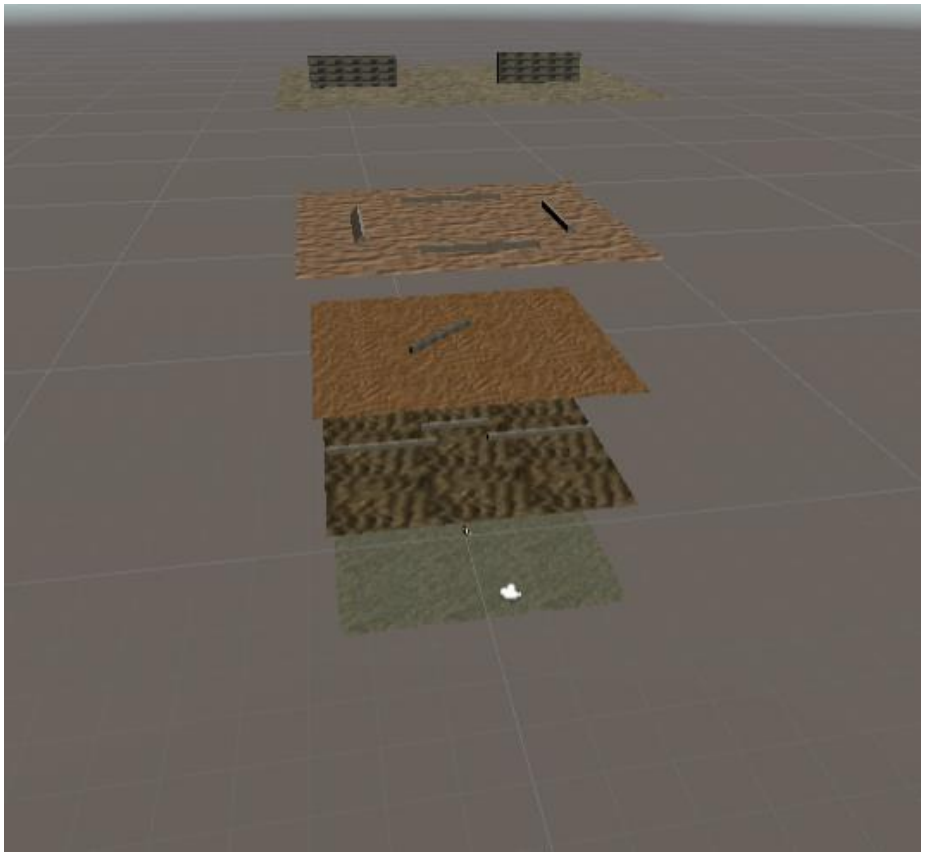

Figura 43: Del piso0 al piso4

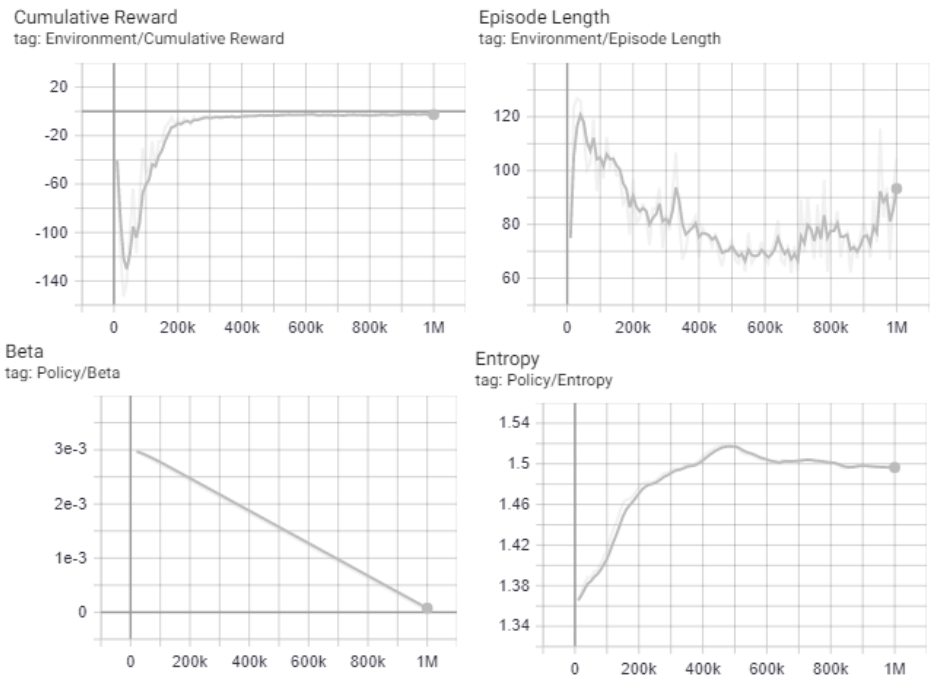

Figura 44: Piso0-piso4 en tensorboard

En esta imagen (figura 44), se puede observar que el entrenamiento tuvo buenos resultados. La recompensa tiene una tendencia ascendente durante el primer cuarto del entrenamiento y seguidamente se estabiliza. La entropía realiza una curva de aprendizaje casi perfecta, pero dentro de los límites esperados, ya que el agente debe aprender la nueva plataforma introducida. En cuanto al tiempo medio en terminar los episodios se puede observar cómo va disminuyendo el tiempo que tarda en acabar los episodios, aunque al final sube un poco. Estas gráficas ponen de manifiesto que el personaje ha conseguido superar la torre entera cogiendo la llave y el cofre sin caerse. Por tanto, los resultados obtenidos son bastante buenos.

4.4 Resultado final

Para comprobar si realmente funcionaba como debía, se utilizó la red neuronal del último entrenamiento como cerebro del personaje. Se cambió el script del agente, de modo que cada vez que caía de una plataforma empezaba otra vez desde la primera plataforma, de esta forma se podía comprobar si el personaje era capaz de completar la torre completa sin caer ninguna vez, y por tanto, se podía comprobar si las recompensas extra que se le implementaron realmente sirvieron para que el agente aprendiese en menos tiempo, pero sin perder de vista el objetivo final.

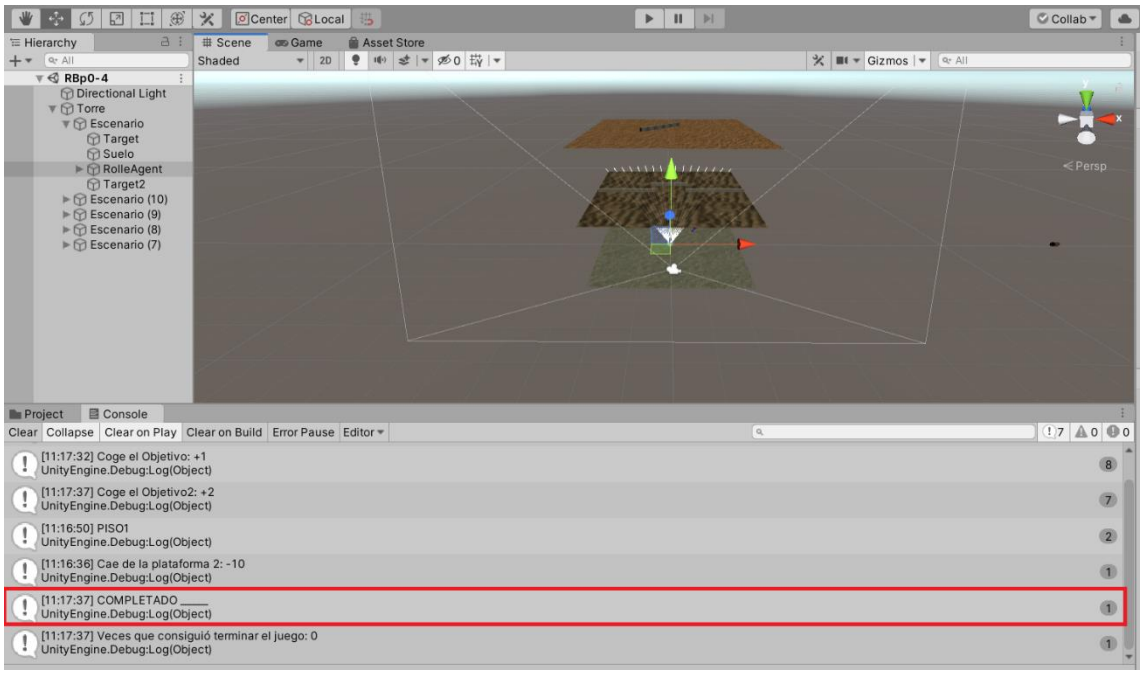

Figura 45: Comprobación final

Una vez realizada la prueba, y como se observa en la imagen (Figura 45), el personaje consigue superar la torre una vez sin caer en aproximadamente un minuto después de su primera caída en el piso 2, por lo que el resultado es muy bueno, pero podría ser aún mejor.

5. Conclusión

En este proyecto se ha presentado como se debe crear y entrenar un entorno en Unity utilizando técnicas de RL. Además se han aportado pruebas de la eficiencia de los entrenamientos en distintos escenarios desarrollados en Unity. Después de todo el trabajo realizado se puede afirmar que en Unity es viable crear un entorno virtual y, con las herramientas de ML-Agents, crear un agente pueda aprender a través de técnicas de aprendizaje por refuerzo. Además, la interfaz de Unity es muy intuitiva y aporta múltiples facilidades, por lo que cuesta poco aprender a trabajar con esta herramienta. Por otro lado, dependiendo de la complejidad de la tarea que desempeña el agente, es necesario establecer distintos parámetros de configuración y utilizar el algoritmo óptimo para su aprendizaje, dependiendo de la dificultad a la que se exponga. En este proyecto, se ha conseguido llegar al objetivo que se propuso en este TFG pero los resultados se pueden mejorar.

5.1 Trabajos futuros

A consecuencia de este proyecto y sus conclusiones, se pueden desarrollar varias mejoras para él mismo, por ejemplo, se podrían animar los movimientos del personaje al moverse o añadir una animación de apertura del cofre cuando el personaje lo coge. También se podrían añadir más pisos con distintas dificultades y mejorar la IA del agente para que realice la tarea sin caerse ni una sola vez. Por otro lado, se podrían añadir obstáculos móviles o que entorpecieran deliberadamente al agente en su tarea.

6. Referencias

[Ward et al., 2020] Tom Ward, Andrew Bolt, Nik Hemmings, Simon Carter, Manuel Sanchez, Ricardo Barreira, Seb Noury, Keith Anderson, Jay Lemmon, Jonathan Coe, Piotr Trochim, Tom Handley and Adrian Bolton. “Using Unity to Help Solve Intelligence.” arXiv 2011.09294 (2020): 1.

[Mnih et al., 2015] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." nature 518.7540 (2015): 529-533.

[Bolt et al., 2020] Andrew Bolt, Tom Ward, Nik Hemmings, Simon Carter, Manuel Sanchez, Ricardo Barreira, Seb Noury, Keith Anderson, Jay Lemmon, Jonathan Coe, Piotr Trochim, Tom Handley and Adrian Bolton. “Using Unity to Help Solve Intelligence.” arXiv 2011.09294 (2020): 2.

[Hemmings et al., 2020] Nik Hemmings, Andrew Bolt, Tom Ward, Simon Carter, Manuel Sanchez, Ricardo Barreira, Seb Noury, Keith Anderson, Jay Lemmon, Jonathan Coe, Piotr Trochim, Tom Handley and Adrian Bolton. “Using Unity to Help Solve Intelligence.” arXiv 2011.09294 (2020): 2.

[Zhang et al., 2020] Junwei Zhang , Zhenghao Zhang , Shuai Han , Shuai Lü. “Proximal Policy Optimization via Enhanced Exploration Efficiency.” arXiv: 2011.05525 (2020): 1.

[Christodoulou et al., 2019] Petros Christodoulou. “Soft Actor-Critic for Discrete Action Settings.” arXiv:1910.07207 (2019) : 1.

Herramienta Tensorboard: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Using-Tensorboard.md

Parámetros de configuración: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md

Unity ML-Agents: https://github.com/Unity-Technologies/ml-agents/

Unity ML-Agents Gym Wrapper: https://github.com/Unity-Technologies/ml-agents/blob/master/gym-unity/README.md