Universidad Politécnica de Cartagena

TRABAJO FIN DE GRADO

Estudio del entorno Unity como herramienta para la definición y prueba de algoritmos de machine-learning para selección de rutas óptimas en conducción autónoma.

Pablo de los Santos Pan

Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicaciones

Grado en Ingeniería Telemática

Índice

Agradecimientos

En especial a mi familia por apoyarme durante toda mi etapa académica y haber hecho posible formarme como ingeniero. A mis compañeros de clase por ayudarme cada vez que lo he necesitado y darme todo su apoyo, también han hecho que todo esto sea posible.

Agradecer también a los profesores durante toda la carrera que me han hecho aprender y formarme de la mejor manera posible, mostrando su apoyo y su ayuda en todo momento.

A mi director por confiar en mí para realizar este proyecto, ayudarme a conseguirlo, por todas sus enseñanzas y sobre todo por su paciencia conmigo.

Sin todos ellos no estaría donde estoy hoy.

RESUMEN

En este proyecto se estudia la viabilidad de la herramienta gráfica Unity para utilizar algoritmos de inteligencia artificial para la selección de rutas óptimas.
Unity cuenta con una librería de machine learning, llamada ml-agents, para la definición de algoritmos de inteligencia artificial. Dicha librería incluye algunos ejemplos ya definidos.
Para entender cómo funciona esta librería, veremos uno de estos ejemplos.

Sin embargo, ninguno de estos ejemplos trata de seleccionar rutas óptimas. Ya que la finalidad del proyecto no es definir un algoritmo propio, sino utilizar algoritmos ya existentes, no se usará esta librería ml-agents, aunque en un futuro proyecto se pretenderá crear un algoritmo de machine learning desde cero.

Estos algoritmos serán utilizados en un entorno urbano, donde un vehículo deberá usar la ruta óptima para llegar a un punto concreto. Inicialmente, deberá decidir la ruta sin ningún otro elemento en la vía urbana, y a lo largo del proyecto se añadirán nuevos elementos como otros vehículos, semáforos, rotondas, etc.

1.- INTRODUCCIÓN

La tecnología avanza a pasos agigantados, con el objetivo de hacernos la vida más fácil. Es evidente que estamos rodeados de ella y la usamos día a día: cuando hablamos por nuestro teléfono móvil, cuando navegamos por internet o al ver la televisión.

En los últimos años está en auge una nueva tecnología, la que podríamos llamar “la tecnología del futuro”: la inteligencia artificial.
La inteligencia artificial es la llamada inteligencia de la máquinas. Una máquina “inteligente” es capaz percibir su entorno y tomar decisiones en función a éste, maximizando la probabilidad de realizar con éxito un objetivo dado.
Aunque aún queda mucho por avanzar en este campo, ya se está empezando a llevar a la práctica: es utilizada en nuestros teléfonos móviles para la detección facial, en asistentes virtuales como Cortana de Microsoft o Siri de Apple. En los videojuegos también es muy usada para dotar de inteligencia a personajes.

Un campo en el que también está teniendo un gran impacto la inteligencia artificial es en el del transporte. Cada vez más empresas de automóviles están trabajando en modelos que poseen este tipo de tecnología como, por ejemplo, la marca Tesla.
Están, así, naciendo los denominados coches autónomos. La implantación de inteligencia artificial en vehículos permitirá evitar accidentes o atascos, así como optimizar el tráfico. Este último punto es en el que nos centraremos en este proyecto, optimizar el tráfico mediante el uso de inteligencia artificial.

En nuestro caso, no vamos a aplicar esta inteligencia en casos prácticos reales como en un coche autónomo, sino que se va a usar la conocida herramienta de desarrollo 3D Unity para probar algoritmos de selección de ruta óptima en un entorno urbano, aplicado a un vehículo.
El coche será el que, mediante inteligencia artificial, decidirá qué camino es óptimo para llegar a un punto concreto.

Para llevar a cabo esta tarea hay varias opciones. Una de ellas es la librería llamada ml-agents, propia de Unity, que viene con algunos ejemplos. Otra opción viable es usar un algoritmo basado en “mallas”. Éste consiste en dividir el terreno en nodos conectados entre ellos, de manera que un agente (así se llama en inteligencia artificial a la “máquina” que posee la inteligencia), nuestro vehículo, calcula el recorrido más corto a través de dichos nodos hasta un objetivo.

2.- CONCEPTOS BÁSICOS

2.1.- Machine Learning

Machine Learning es una rama de la inteligencia artificial que consiste en crear sistemas capaces de aprender por sí mismos. Las decisiones se toman en base a datos recogidos del entorno.
Pretende estudiar el reconocimiento de patrones y el aprendizaje de máquinas.
Según **Arthur Samuel**, pionero en el campo de la inteligencia artificial y creador de uno de los primeros programas de “auto-aprendizaje” del mundo, este campo de la ciencia le da a las máquinas la capacidad de aprender sin ser programadas explícitamente.

Dicho de otra manera, son algoritmos que proporcionan conclusiones relevantes a partir de un conjunto de datos obtenido, sin necesidad de escribir instrucciones específicos para esto.
Un algoritmo de Machine Learning está compuesto por unos datos recibidos por la máquina, una función de coste que se pretende minimizar, una función de optimización a maximizar y un modelo.

2.2.- Tipos de Machine Learning.

Un sistema informático de aprendizaje automático requiere unos datos con los que aprender unos patrones o comportamientos.
A partir de estos datos, existen tres tipos principales de Machine Learning: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.

2.2.1.- Aprendizaje supervisado.

En este tipo de entrenamiento, se entrena al sistema proporcionándole unos datos etiquetados, definidos detalladamente. Por ejemplo, proporcionándole unas fotos de perros y de gatos con etiquetas definiéndolas como tales.

Una vez que se le ha proporcionado una cantidad suficiente de datos, se pasará a introducir nuevos datos sin etiquetas, de manera que el sistema, siguiendo los patrones que ha ido registrando durante el entrenamiento, los etiquetará.
Este sistema se conoce como **clasificación**.

Algunos ejemplos de este tipo de aprendizaje son: detección de spam en correos electrónicos, detección de imágenes en captchas, reconocimiento de escritura o de voz, etc.

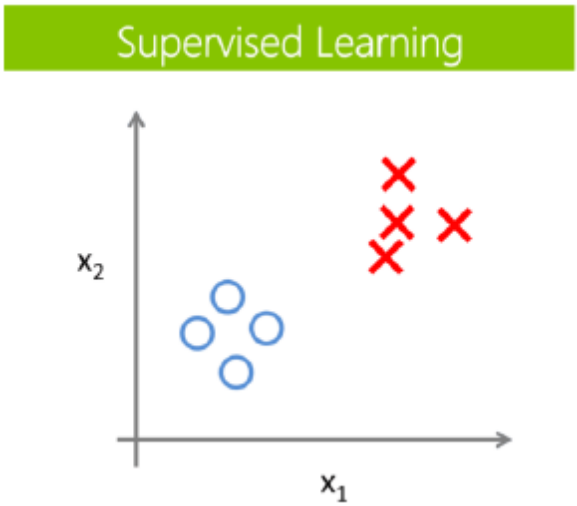

Figura 2.1. Ejemplo de aprendizaje supervisado (clasificación).

En cambio, si el resultado deseado consiste en predecir un valor continuo dado unos parámetros distintos, el sistema se conoce como **regresión**.

Este tipo de aprendizaje se utiliza, muy comúnmente, para predecir el valor de una vivienda o un terreno, como se muestra en la siguiente figura:

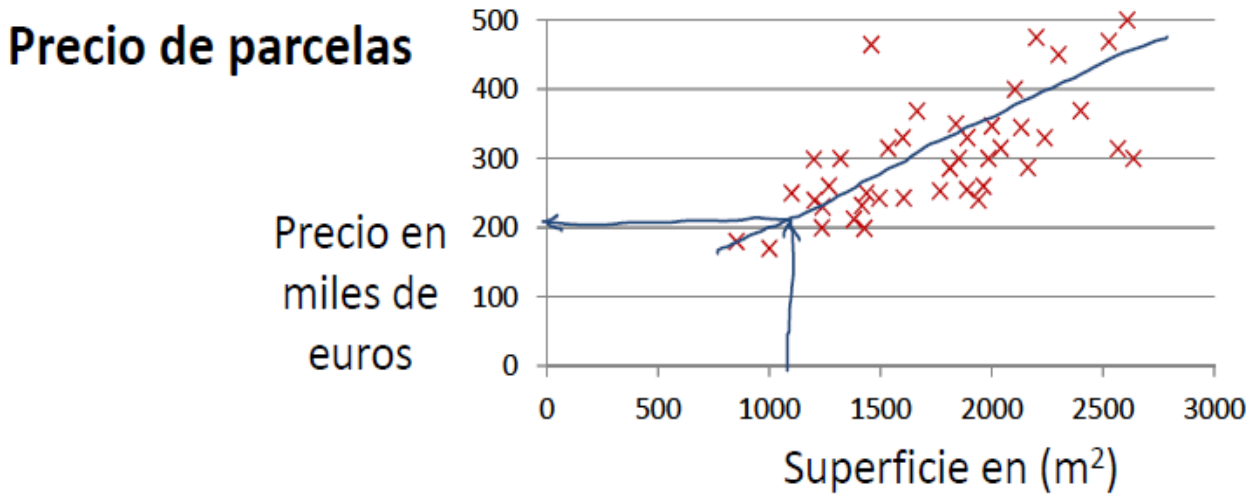

Figura 2.2. Ejemplo de regresión en aprendizaje supervisado.

2.2.2.- Aprendizaje no supervisado.

En este caso, los algoritmos no necesitan datos etiquetados para su aprendizaje, de manera que no cuenta con ninguna indicación previa, sino que se les otorgan las características. Sólo se dispone de datos de entrada, pero no hay datos de salida correspondientes a una determinada entrada.
Su función es la de agrupar los datos en función a las características de éstos, pero sin distinguir individualmente a cada uno.
Estos grupos en los que se clasifican los datos se llaman **clusters**.

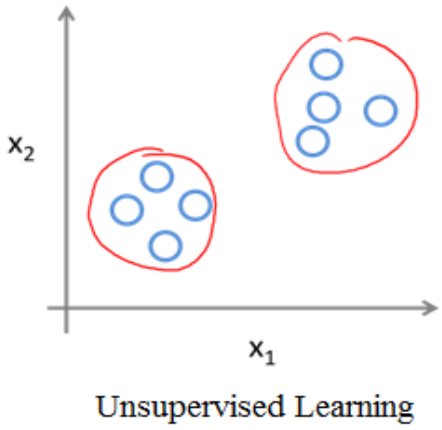

Figura 2.3. Ejemplo de clustering con dos clusters.

2.2.3.- Aprendizaje por refuerzo.

El aprendizaje por refuerzo se basa en la **psicología conductista** del ser humano, que consiste en que una acción concreta conlleva un resultado deseado haciendo más probable que el sujeto repita esta acción, o un resultado no deseado que significa una disuasión del sujeto a realizar de nuevo dicha acción. Por ese motivo, con este tipo de aprendizaje, las máquinas pueden tomar decisiones aunque no almacenen previamente conocimientos o variables del entorno, como sí sucedía con el aprendizaje supervisado.
Por ejemplo, nos satisface sacar notas altas o jugar a un videojuego, por lo que estudiamos/jugamos más horas para satisfaces esos estímulos positivos (reforzamientos) y probamos nuevos procesos de **prueba y error** para evaluar nuevas estrategias de estudio/juego que tengan un mayor resultado satisfactorio.

Se basa en aplicar este aprendizaje del ser humano a una inteligencia artificial (máquinas), con el fin de que aprendan por ellas mismas. Además, las máquinas poseen dos características de las que carecemos los seres humanos: no se cansan ni se aburren y realizan las tareas a una velocidad infinitamente más rápida que nosotros.
Para el ejemplo de los videojuegos, una máquina puede ganar o perder millones de partidas en un videojuego en poco tiempo, llevándola a saber qué decisiones le llevaron a la victoria o a la derrota, de manera que prioriza unas decisiones y descarta otras hasta que la estrategia sea perfecta y siempre gane.
Aplicando este principio, una máquina puede llegar a realizar cualquier tipo de tarea, siempre y cuando pueda recibir **castigos y recompensas** por las decisiones que toma en cada una de las iteraciones de prueba y error.

El aprendizaje por refuerzo es la clave para que **la conducción autónoma sea una realidad**: que un vehículo sea capaz de incorporarse a una carretera, que sepa manejar una situación de atasco o, como es el tema de este proyecto, **decidir una ruta óptima hasta un punto concreto**.

Pero, ¿qué se necesita para formular un problema de aprendizaje por refuerzo?
El aprendizaje por refuerzo consta de un **agente** como por ejemplo un perro, dentro de un **entorno** del que recoge la información como puede ser el dueño dándole órdenes, en un **estado** determinado, bien en reposo o bien realizando alguna orden. El perro recibe una **recompensa** en forma de comida cada vez que realiza bien una orden y un **castigo** cuando la realiza mal u otra orden distinta, es decir, dependiendo de la **acción** que realice.

Podemos suponer que el aprendizaje sigue un **proceso de decisión de Markov**, por lo que se podría decir que:
- El agente percibe un conjunto finito S de posibles de estados en el entorno y dispone de un conjunto finito A de acciones a realizar.
- Para cada instante discreto de tiempo t, el agente percibe un estado s_t y realiza una acción a_t, obteniendo un nuevo estado $s_{t+1} = a_t(s_t)$.
- El entorno responde a la acción tomada por el agente en forma de recompensa o castigo $r(s_t, a_t)$.
- El agente no sabe cuál será el estado siguiente al realizar la acción tomada, ya que un buen aprendizaje le permite ser capaz de adelantarse a los consecuencias de dichas acciones tomadas, de manera que el agente aprende qué acciones y sobre qué estados concretos le llevan a conseguir un mayor beneficio.

La siguiente figura muestra un esquema del funcionamiento cíclico del aprendizaje por refuerzo.

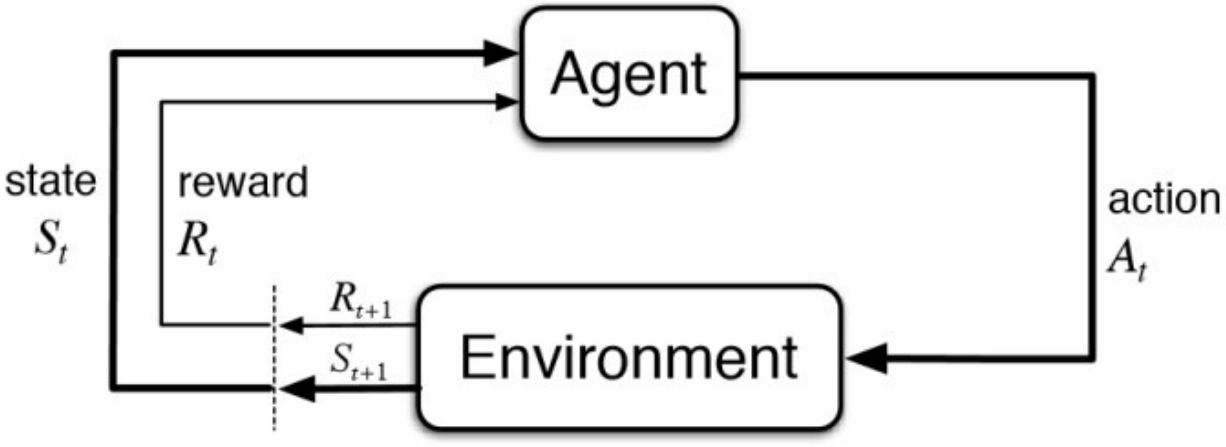

Figura 2.4. Funcionamiento del aprendizaje por refuerzo

El objetivo final es el de maximizar la **recompensa acumulada esperada**, que no es otra cosa que la suma de todas las recompensas que ha ido obteniendo el agente durante el entrenamiento. Por lo tanto, la función de maximización se puede expresar como:

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T$$

donde R_t es la recompensa total y T es el estado final. Los estados finales indican que el agente ha terminado el episodio. Una vez se le ha asignado una recompensa, el proceso se reinicia, buscando mejorar la recompensa acumulada esperada, es decir, la recompensa total.

La librería de Unity ml-agents usa más comúnmente este tipo de aprendizaje para entrenar a los agentes. El funcionamiento se explicará más adelante.

2.3.- Redes Neuronales

Las redes neuronales imitan, como su nombre indica, las funciones de las redes neuronales de los organismos vivos. Es decir, son un conjunto de “neuronas” conectadas entre sí, de manera que, mediante un input, las neuronas trabajan para devolver un input a partir de haber “aprendido”.

Las redes neuronales poseen una capa de entrada, una o más capas ocultas que son las que se encargan del procesamiento, y una capa de salida, la cual se encargará de devover respuestas a las entradas.

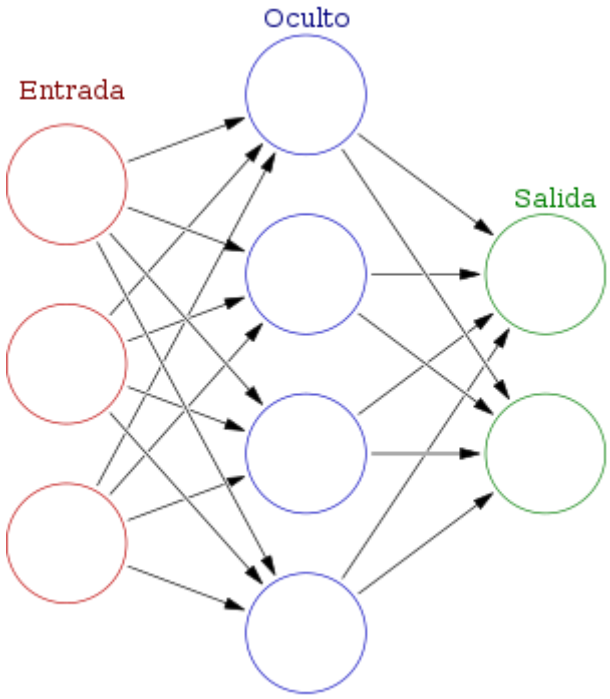

Figura 2.5. Esquema básico de una red neuronal

Las redes neuronales son entrenadas para que, para unas mismas entradas, devuelvan salidas cada vez más semejantes a lo que se espera de ellas. Es decir, como nuestro cerebro.

3.- HERRAMIENTAS

En esta sección introduciremos las herramientas que se han utilizado en este proyecto.

3.1.- Unity

Unity es un motor para desarrollar videojuegos creado por la empresa danesa Unity Technologies en 2005.

Es compatible con muchos otros software de diseño como Blender, Maya, Adobe Photoshop o 3ds Max, que son usados para crear modelos 3D e importarlos a Unity sin tener que realizar ninguna transformación.

Está desarrollado en C++ y permite el uso del lenguaje de programación C# principalmente para crear los videojuegos.

Sin duda, unas de sus grandes ventajas es su simpleza y facilidad de utilizar, con multitud de funciones de “arrastrar y pegar” como por ejemplo a la hora de añadir un modelo a la escena o asociar un script a un objeto, una pantalla que muestra lo que el jugador ve dentro del videojuego, otra pantalla en la que trabajar el programador, un sección con los parámetros y configuraciones de cada objeto dentro de la escena y otras muchas más funcionalidades que hacen de Unity una herramienta fácil de usar.
De esta manera, permite que un principiante pueda empezar a usar esta herramienta sin necesidad de tener gran conocimiento de programación de videojuegos.

Permite también crear tanto videojuegos en 2D como en 3D con una amplia librería de *físicas* para su fácil utilización.
Cabe mencionar un facilidad para manejar las cámaras, pudiendo crear varias en una misma escena e ir cambiando entre ellas fácilmente. Dispone también de diferentes tipos de iluminación para crear la luz adecuada para cada situación, así como parámetros que son fácilmente modificables como el color o el brillo de las luces.

Para la programación, se usan scripts, que son archivos independientes que se asignan a los objetos del juego. Estos archivos sirven para dotar a los objetos de una funcionalidad, descrita en dicho archivo, como puede ser el movimiento de un personaje por el escenario.

En definitiva, la interfaz gráfica de Unity es muy visual, intuitivo y fácil de comprender, con la capacidad además de repartir cada una de las ventanas como al programador le sea más cómodo.

En la figura 3.1 se muestra la interfaz de Unity, destacando cada uno de sus componentes.

- En la ventana de jerarquía (hierarchy window) se muestran todos los objetos que hay en la escena, pudiéndolos seleccionar, eliminar, etc.

- El inspector muestra todos los componentes que tiene el objeto seleccionado (ya sea en la escena o en la ventana de jerarquía), así como los parámetros, que el programador podrá modificar. También muestran los scripts que tienen asociados.

- La ventana de proyecto muestra todas las carpetas y archivos existentes dentro de la carpeta local del proyecto. Podemos crear escenas nuevas, añadir prefabs a la escena y navegar por todas las carpetas del proyecto. Un prefab es un objeto configurado y almacenado dentro de la carpeta del proyecto, con todos sus componentes y scripts ya asociados.

- En el centro de la interfaz aparecen dos pantallas: la vista de la escena y la vista del juego. En la vista de la escena será donde el programador creará y modificará el juego, colocará los objetos, etc. En definitiva, será donde trabaje. Mientras tanto, en la vista del juego se muestra lo que el jugador verá una vez iniciado el juego. En esta pantalla no se puede trabajar.

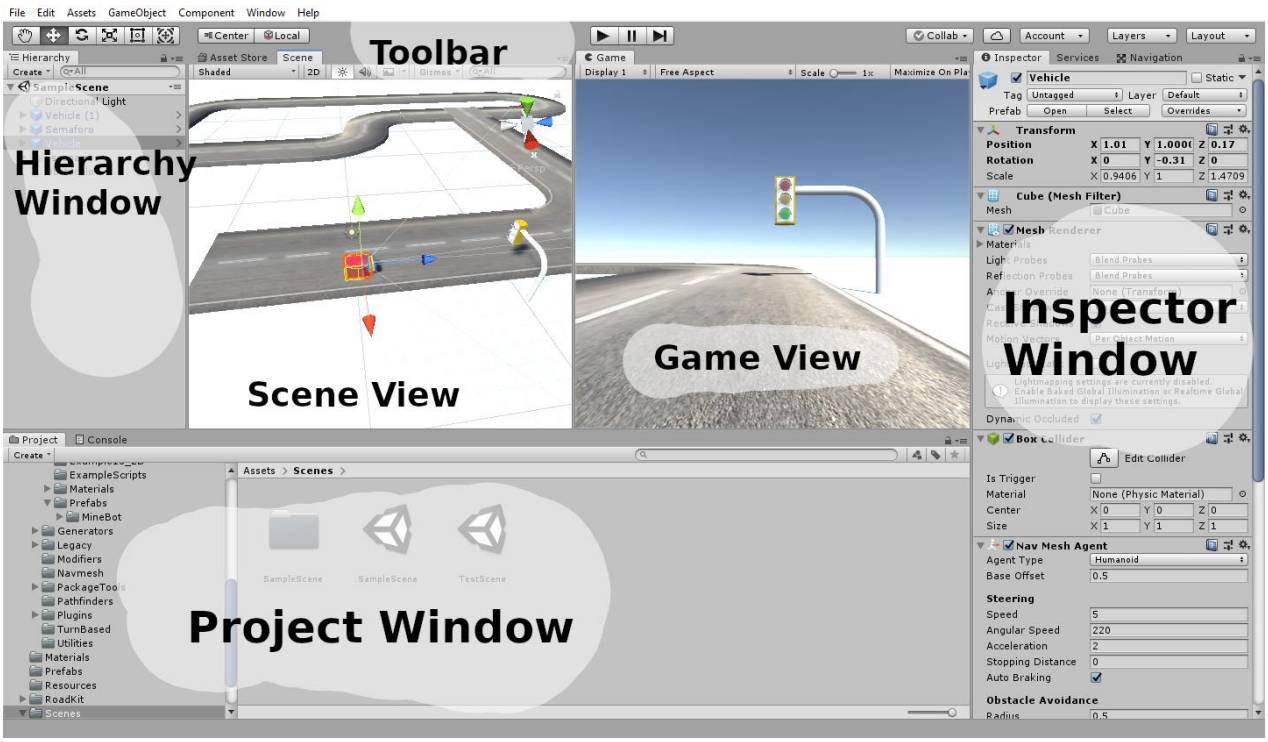

Figura 3.1. Interfaz gráfica de Unity

3.2.- TensorFlow

TensorFlow es una librería de código abierto escrita en Python y C++ desarrollado por Google y que sirve para dotar de inteligencia artificial a los agentes de Unity, entre otras muchas cosas.

Cuenta con una API escrita en Python con la que se pueden entrenar a los agentes mediante diferentes algoritmos.

Para entrenar diferentes modelos es necesario especificar unos parámetros que requiere el algoritmo a utilizar, llamados hiperparámetros, que son imprescindibles para un correcto entrenamiento. Estos hiperparámetros se deben ajustar hasta conseguir un entrenamiento adecuado, y encontrar los valores correctos requiere varias iteraciones.
Para ello, TensorFlow ofrece una herramienta para visualizar ciertos atributos del agente (como por ejemplo la recompensa) a lo largo del entrenamiento, lo que resulta útil para establecer valores óptimos para los hiperparámetros.

TensorFlow cuenta además con una utilidad llamada **Tensorboard** que nos permite ver y guardar estadísticas durante la sesión de aprendizaje.

3.3.- Librería de Machine Learning de Unity

Unity cuenta con una librería propia para el entrenamiento de agentes mediante machine learning llamada **ml-agents**.
Esta librería permite al desarrollador transformar juegos y simulaciones en entornos donde entrenar agentes inteligentes usando métodos de aprendizaje máquina.

Para poder utilizar esta librería, debe importarse como un asset de Unity. Un asset es un archivo importado que sirve para ser utilizado dentro de Unity, como puede ser un modelo 3D creado en Blender, un archivo de audio, una imagen o cualquier otro tipo de archivo que soporta Unity. También puede ser creado dentro del propio Unity como un Animator Controller, un Audio Mixer o una Render Texture.
Concretamente la librería de ml-agents es un asset que está formado por una estructura de directorios y ficheros, como se muestra en la figura 3.2.

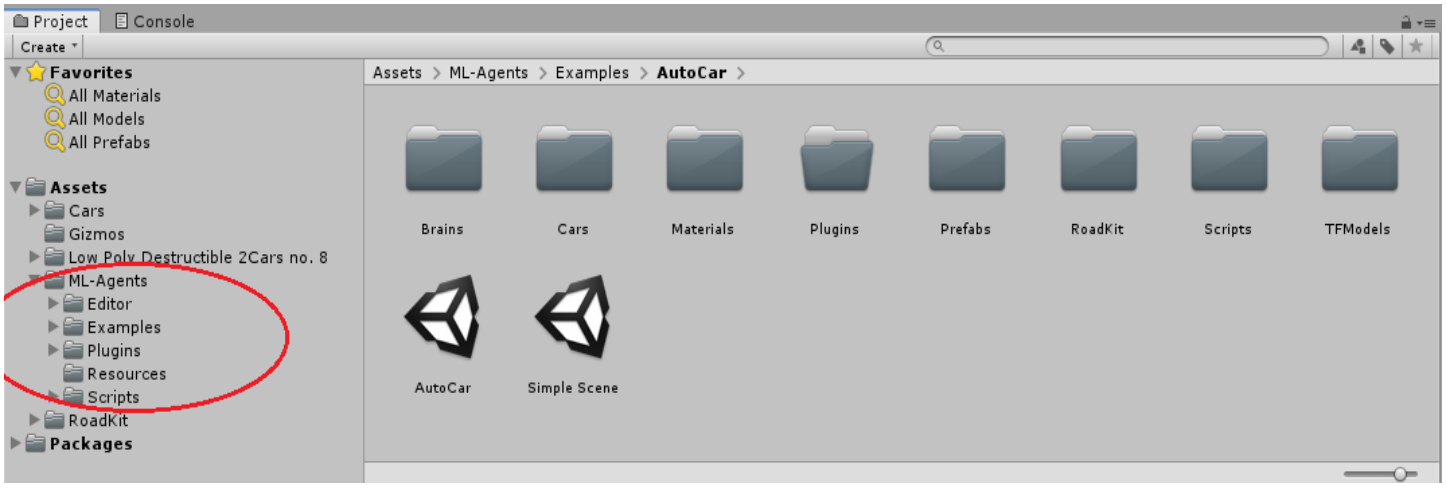

Figura 3.2. Estructura de directorios y ficheros de la librería ml-agents.

En la carpeta *Plugins* se encuentra la librería de TensorFlow, con el que se entrenan a los agentes. La carpeta *Editor* contiene parámetros modificables del comportamiento del cerebro, que es el objeto que proporciona inteligencia

al agente). En la carpeta *Examples* se encuentran algunos ejemplos de algunos proyectos de ml-agents para comprender su funcionamiento.

La carpeta *Scripts* contiene todos los archivos necesarios para que la librería funcione y un objeto pueda aprender. Son de código abierto y modificables aunque no se recomienda hacerlo para no alterar el funcionamiento. Cada script representa una clase, con sus respectivos métodos.
Dentro de esta carpeta, los scripts más importantes y que se utilizarán con más frecuencia son los siguientes:

- **Agent:** clase que deben extender todos los scripts asociados a los objetos que se quieren convertir en agentes que aprenderán dentro del entorno.
  El agente recibe la información del entorno, realiza una acción y recibe una recompensa dependiendo de la situación y la acción que haya tomado.
  La decisión que toma depende del **cerebro (brain)**, que es el encargado de recibir la información del agente, procesarla, y enviarle la acción que debe realizar.
  Un agente solamente puede tener un cerebro, aunque un mismo cerebro puede estar asociado a más de un agente.
  Las funciones más importantes que implementa esta clase son:

1.- **Start()**: donde se define el valor inicial de todos los parámetros y variables del agente. Se ejecutará una única vez cuando se ejecute el juego o simuación.

2.- **AgentAction(float[] vectorAction, string textAction)**: función encargada de realizar la acción del agente. El parámetro *vectorAction* es un vector de tipo float que recibe la acción a realizar del cerebro. Esta acción se procesa y se recompensa o castiga al agente dependiendo de la situación. Dicha recompensa se añade mediante la función AddReward(float increment), donde el parámetro *increment* puede ser un valor positivo o negativo, dependiendo de si se recompensa o se castiga al agente.

3.- **CollectObservations()**: esta es la función que se encarga de recoger todos los datos del entorno para que el cerebro tenga un contexto para realizar todos los cálculos necesarios a la hora de tomar una decisión de qué acción realizar por parte del agente. Para añadir los datos del entorno se usa la función *AddVectorObs()*, que recibe como parámetro cualquier tipo numérico, como un vector de tres posiciones indicando la posición en el espacio de un determinado objeto, o la velocidad de un coche dentro del juego o simulación.

4.- **AgentReset()**: esta función se ejecuta cada vez que termina una iteración del entrenamiento, reiniciando los valores y parámetros que se hayan modificado durante dicha iteración, como puede ser la posición del agente.

5.- **Done()**: esta función finaliza una iteración del entrenamiento. Al ejecutarse, también se ejecuta la función AgentReset(), ya que ha terminado una iteración.

- **Brain:** al igual que en el caso del ser humano, es el encargado de procesar toda la información recibida del entorno por el agente, procesarla y decidir qué decisión tomar. Cada cerebro define un estado específico y un espacio de acción.
Actualmente existen cuatro tipos de cerebros:

1.- **External**: las decisiones se toman usando el algoritmo PPO (Proximal Policy Optimization) que proporciona la API de TensorFlow. La información se envía a la API a través de una comunicación socket utilizando un *external communicator*.

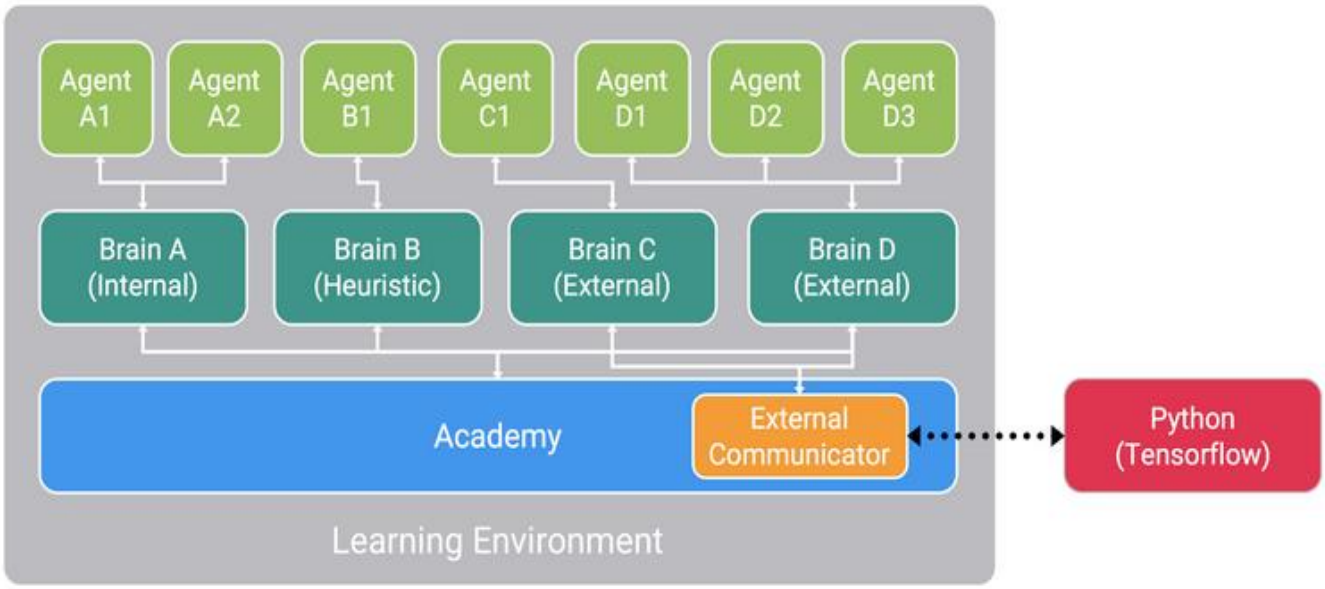

Figura 3.3. Esquema de funcionamiento de ML-Agents

2.- **Internal**: las decisiones se toman en función a la política generada por la API de TensorFlow al entrenar al agente con un *External Brain*. Es decir, se usa un modelo ya entrenado. No necesita ningún tipo de comunicación.

3.- **Player**: las decisiones las toma un jugador por medio de un controlador como el teclado. Se utiliza para comprobar el correcto funcionamiento de las recompensas. El cerebro **no aprende**.

4.- **Heuristic**: las decisiones se toman atendiendo a un código escrito por el programador. Se suele utilizar para comprobar que el entrenamiento por parte de la API mejora el del código escrito.

- **Academy**: El objeto Academy es una colección de objetos Brain. Cada agente tiene asociado un único cerebro, aunque un cerebro puede estar asociado a más de un agente.
Este objeto Academy contiene el *External Communicator* que se comunica con la API de TensorFlow cuando el cerebro está en modo *external*.

Todos los estados y las observaciones de los agentes con cerebro establecido en *External* son recogidos por el *External Communicator* que se comunica con la API elegida, en nuestro caso, TensorFlow.

Asociando varios agentes a un mismo cerebro conseguimos realizar un entrenamiento paralelo, acelerando en gran medida la velocidad de aprendizaje.

3.4.- Anaconda

Anaconda es un software que nos permite manejar entornos separados para diferentes distribuciones de Python. Con esta herramienta se ha creado un entorno enfocado a machine learning llamado ml-agents.
Se utiliza además para instalar dentro de este entorno la librería de TensorFlow que permite entrenar a los agentes con diferentes algoritmos de machine learning.
En otras palabras, en este programa será donde se realicen los entrenamientos.

4.- LIBRERÍA ML-AGENTS DE UNITY

En esta sección veremos cómo funciona la librería de inteligencia artificial ml-agents de Unity. También veremos un ejemplo de los que vienen incluidos en la propia librería, para comprender su funcionamiento.

4.1.- Ejemplo: Pelota en equilibrio

El objetivo de este ejemplo es conocer y aprender las posibilidades que permite este software.

El ejemplo típico que ofrece Unity con su librería es el de una pelota en equilibrio sobre una plataforma.
La pelota cae a una plataforma desde una cierta altura y la plataforma, que es el agente, deberá rotar en el eje X y el eje Z para conseguir que la pelota no caiga de dicha plataforma. Se utiliza el aprendizaje por refuerzo anteriormente explicado, en el que se castiga a la plataforma cada vez que la pelota cae.

La escena está compuesta por dos únicos elementos: una esfera, que será a pelota que se debe mantener en equilibrio, y un plano, que representará la plataforma que debe evitar que la pelota caiga.

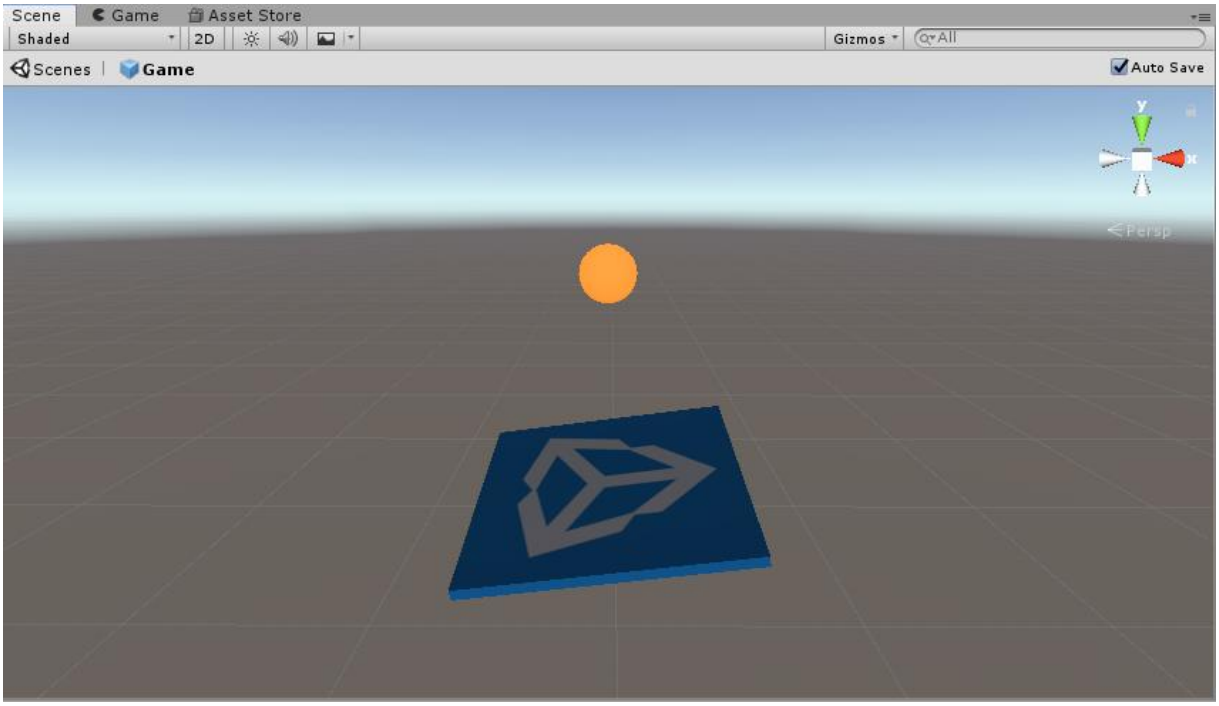

Figura 4.1. Plataforma (agente) y pelota del ejemplo de ml-agents

4.1.1. Recompensa y castigos

El aprendizaje por refuerzo, como se ha explicado anteriormente, se basan en tomar decisiones en base a las observaciones del entorno del agente. Dichas decisiones se premian o se castigan dependiendo de qué haya sucedido al llevarla a cabo. Por eso, la función de recompensa es la parte más importante a la hora de definir un algoritmo de aprendizaje por refuerzo como es en el caso del ejemplo.

El algoritmo PPO que proporciona, en nuestro caso, la API de TensorFlow recibe como parámetros de entrada las observaciones del entorno, que son recogidas en la función *CollectObservations()* mediante la función *AddVectorObs()* en el script del agente, que deberá extender a la clase *Agent* de la API de ml-agents.
Estos parámetros serán los inputs para la red neuronal.

En este ejemplo de la pelota en equilibrio, las observaciones recogidas del entorno, como se puede apreciar en la figura 4.2, son las siguientes:

- La rotación de la plataforma en el eje X y en el eje Z, para que la red neuronal sepa en todo momento la rotación de la plataforma.
- La posición de la pelota relativa a la plataforma, ya que a la posición real de la pelota se le resta la posición de la plataforma. Esta “observación” es un vector de tres unidades, una por cada eje, ya que la posición de un objeto es en el espacio de tres dimensiones.
- Por último, la velocidad de la pelota, que también se representa como un vector de tres posiciones.

```
public override void CollectObservations()
{
    AddVectorObs(gameObject.transform.rotation.z);
    AddVectorObs(gameObject.transform.rotation.x);
    AddVectorObs(ball.transform.position - gameObject.transform.position);
    AddVectorObs(ballRb.velocity);
}
```

Figura 4.2. Función de observaciones del entorno

En total, suman 8 observaciones: la rotación del eje X, la rotación del eje Z, el vector de 3 unidades correspondiente a la posición de la pelota y el vector de tres unidades correspondiente a la velocidad de la pelota.
Estas 8 observaciones serán los parámetros de entrada de la red neuronal, que formarán la input layer (capa de entrada).

A continuación, el agente deberá tomar una decisión en función a dichas observaciones. Como ya se ha visto, la función encargada de llevar a cabo la toma de decisiones es la función *AgentAction(float[] vetorAction, string textAction).*
Dicha función se encarga de, dependiendo de la decisión tomada, rotar la plataforma, tanto en el eje X como en el eje Z. Estas dos salidas forman la llamada output layer (capa de salida).

Una vez que se ha realizado la decisión tomada, se debe valorar si dicha acción es la acción correcta, es decir, si el agente ha hecho lo que se esperaba de él. Para ello, se usa la función de recompensa.

En la misma función de la realización de la decisión tomada, una vez se ha realizado dicha acción, se pasa a comprobar el estado de la pelota. Si la pelota se ha caído de la plataforma, esto es, su posición con respecto a la plataforma es negativa (concretamente inferior a 2) o su posición en X o en Z sobrepasa el borde de la plataforma, se establece una recompensa negativa de -1.
En cambio, si no se cumple ninguna de estas condiciones, es decir, la pelota sigue en la plataforma después de haber sido rotada, se establece una recompensa positiva de 0.1.
Como se puede observar, es mucho mayor la recompensa negativa que la positiva. De esta manera, se consigue que el algoritmo “se dé cuenta” mucho antes de las acciones negativas y que, por lo tanto, las intente evitar antes, acelerando el proceso de aprendizaje.

```
public override void AgentAction(float[] vectorAction, string textAction)
{

    if (brain.brainParameters.vectorActionSpaceType == SpaceType.continuous)
    {
        var actionZ = 2f * Mathf.Clamp(vectorAction[0], -1f, 1f);
        var actionX = 2f * Mathf.Clamp(vectorAction[1], -1f, 1f);

        if ((gameObject.transform.rotation.z < 0.25f && actionZ > 0f) ||
            (gameObject.transform.rotation.z > -0.25f && actionZ < 0f))
        {
            gameObject.transform.Rotate(new Vector3(0, 0, 1), actionZ);
        }

        if ((gameObject.transform.rotation.x < 0.25f && actionX > 0f) ||
            (gameObject.transform.rotation.x > -0.25f && actionX < 0f))
        {
            gameObject.transform.Rotate(new Vector3(1, 0, 0), actionX);
        }
    }
    if ((ball.transform.position.y - gameObject.transform.position.y) < -2f ||
        Mathf.Abs(ball.transform.position.x - gameObject.transform.position.x) > 3f ||
        Mathf.Abs(ball.transform.position.z - gameObject.transform.position.z) > 3f)
    {
        Done();
        SetReward(-1f);
    }
    else
    {
        SetReward(0.1f);
    }
}
```

Figura 4.3. Función de realización de la acción y posterior recompensa

Con estas recompensas, la red neuronal será capaz de ir modificando la política a lo largo de las iteraciones del aprendizaje con el objetivo de optimizarla, es decir, conseguir la mayor recompensa posible.

4.1.2. Entrenando el modelo

Una vez el entorno está preparado y se han diseñado las funciones de recompensa y recogida de observaciones del entorno, es hora de entrenar el modelo, es decir, aplicar el aprendizaje del agente para que éste aprenda. En este caso, que aprenda a mantener la pelota en equilibrio.

Como ya hemos visto anteriormente, un único cerebro se puede asociar a varios agentes para agilizar el entrenamiento. Esto es lo que se ha hecho en este ejemplo. Para un mismo cerebro hay 12 plataformas con sus respectivas pelotas aprendiendo al mismo tiempo, como se puede observar en la figura 4.4. De esta manera, cuando a una plataforma se le caiga la pelota y reciba una recompensa negativa, el cerebro será consciente de esto y, como todas las demás plataformas están asociadas al mismo cerebro, éstas también habrán aprendido que si su pelota cae, recibirán una recompensa negativa o castigo.

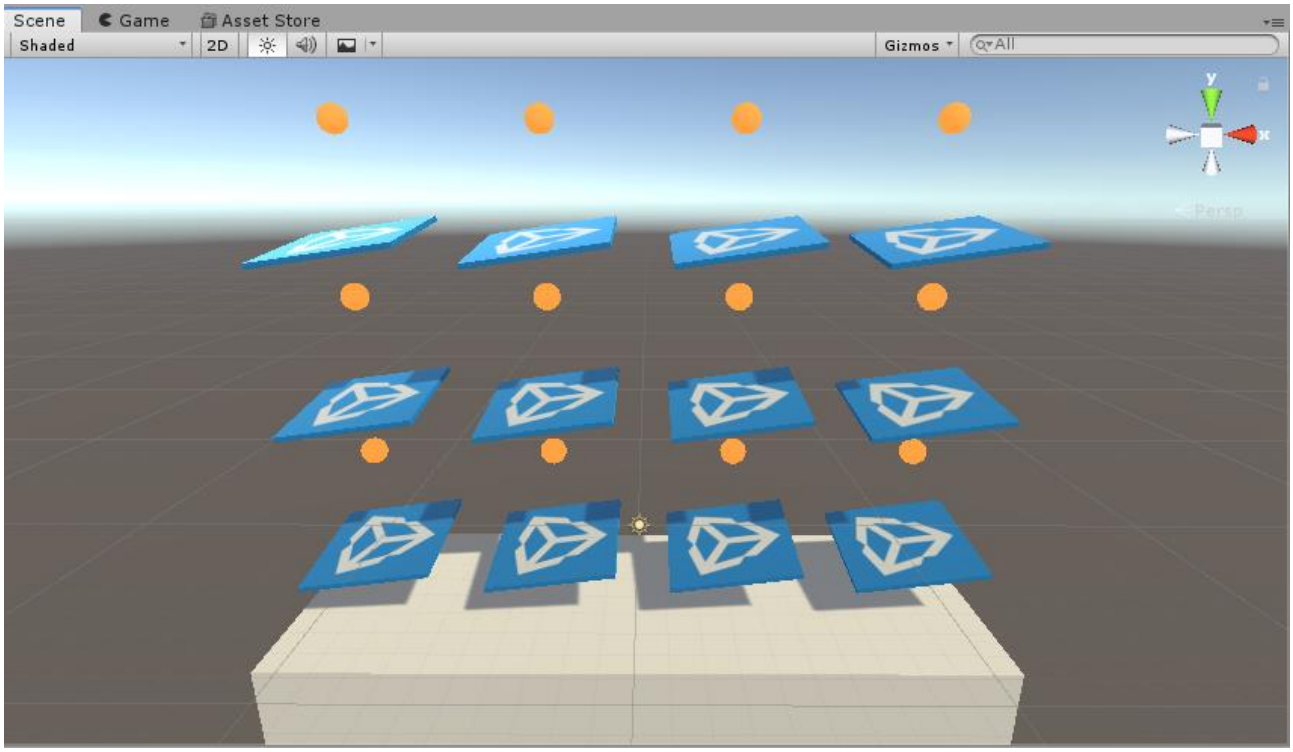

Figura 4.4. Escenario del ejemplo. Las doce plataformas y sus pelotas

Utilizaremos el software Anaconda y el entorno ml-agents creado para llevar a cabo el entrenamiento. Antes de comenzar el entrenamiento, se muestran en la consola de comandos de Anaconda (ya que carece de interfaz gráfica) los *hiperparámetros* ya comentados. Nosotros no los hemos tocado, ya que es un entrenamiento bastante básico y no es necesario cambiarlos. A lo largo del entrenamiento se van mostrando en dicha consola de comandos los pasos que se van completando con alguna información como el tiempo transcurrido o, más importante, la recompensa media en ese momento del entrenamiento.

A continuación se muestra una captura de pantalla de algunas de las líneas que se han mostrado al entrenar a las plataformas en nuestro ejemplo:

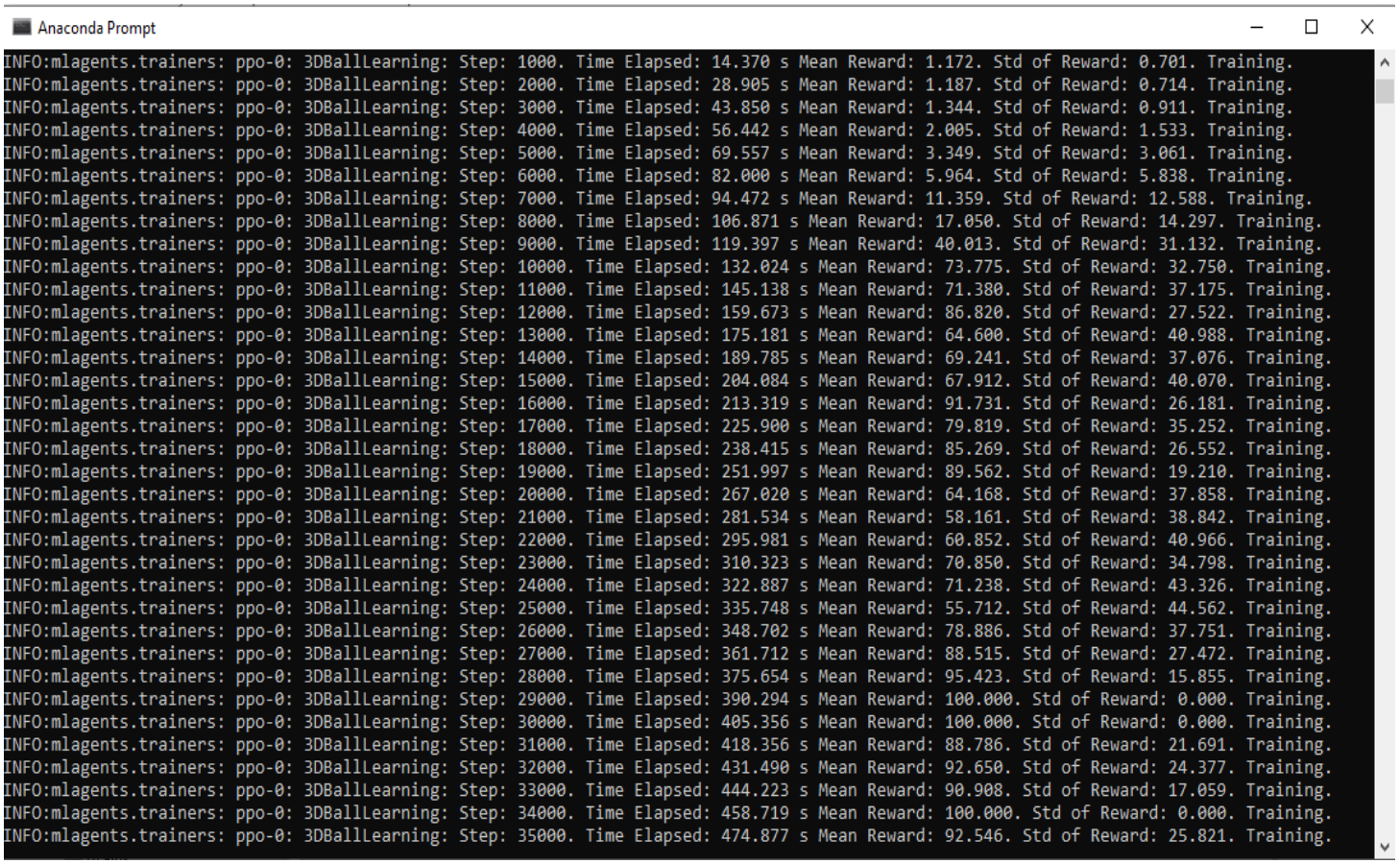

Figura 4.5. Salida de la consola de comandos de Anaconda durante el entrenamiento

Se puede apreciar claramente cómo la recompensa media (Mean Reward) empieza valiendo un número bastante bajo (en torno a 1) y conforme va avanzando el entrenamiento va aumentando hasta quedarse en torno a 100. Aquí se demuestra cómo los agentes, en nuestro caso las plataformas, han ido poco a poco aprendiendo y maximizando la recompensa, es decir, optimizando la política.
Se puede observar también que, como es un entrenamiento bastante sencillo, no se han necesitado un gran número de pasos o iteraciones para llegar al valor de recompensa óptimo. Con unas 35.000 o 40.000 iteraciones ha sido más que suficiente.

Como se ha mencionado anteriormente, TensorFlow cuenta con una utilidad llamada tensorboard para visualizar las estadísticas del aprendizaje. En la siguiente figura se muestran sólo algunas de las gráficas que se pueden visualizar con tensorboard, correspondientes al aprendizaje de las plataformas.

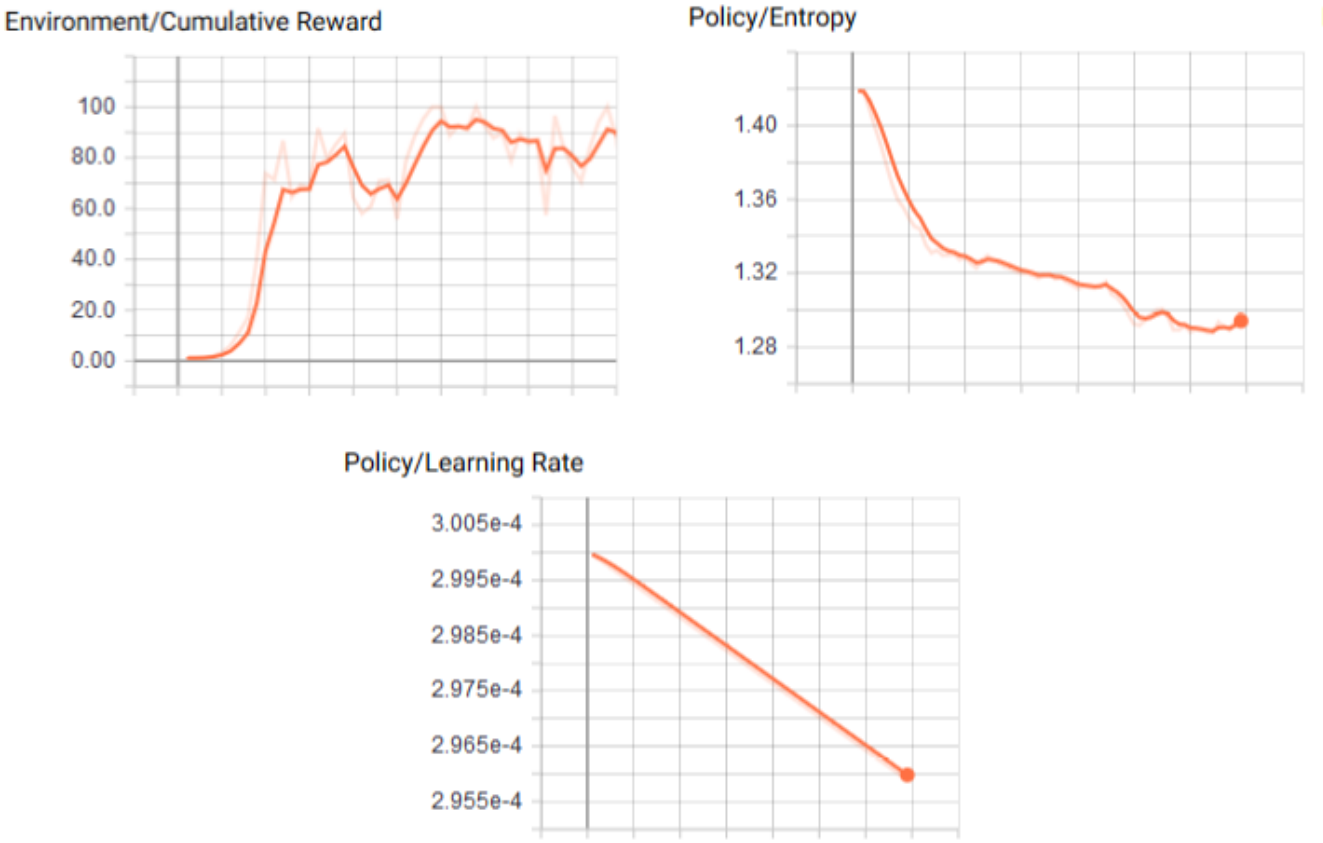

Figura 4.6. Gráficas del entrenamiento en tensorboard

Se puede observar claramente, una vez estudiadas las gráficas, que el aprendizaje avanza con más rapidez al principio, es decir, se aprende más al principio del entrenamiento.

En la gráfica que muestra la recompensa acumulada (cumulative reward) se aprecia que durante los primeros instantes del entrenamiento la subida es bastante pronunciada, y conforme avanza, sube con mucha menos claridad, quedando más constante.

La entropía (entropy) indica la aleatoriedad de las acciones que toma el agente. Se observa cómo al empezar el entrenamiento hay una gran aleatoriedad en la toma de decisiones, pero rápidamente va aprendiendo y no tarda en disminuir la aleatoriedad de dichas acciones. Esto quiere decir que, como ya ha aprendido de las acciones que ha tomado al principio del entrenamiento, el cerebro empieza a tomar más decisiones conforme a dicho aprendizaje, en lugar de ser tan aleatorias.

La última gráfica muestra la velocidad del aprendizaje (learning rate). Muestra un claro descenso, lo que indica que al principio es cuando más aprende, y conforme va avanzando el entrenamiento, el aprendizaje es más pequeño.

Así, se ha demostrado que cuando más aprende el cerebro es al principio del entrenamiento.

Por último, este entrenamiento genera un archivo de tipo .nn que contiene el modelo ya entrenado, como se puede apreciar en la figura 4.7. Este archivo se asociará al cerebro dentro de Unity para que, en lugar de comunicarse con TensorFlow para entrenar, los agentes utilicen este modelo para comportarse tal y como han aprendido en el entrenamiento que generó el archivo.

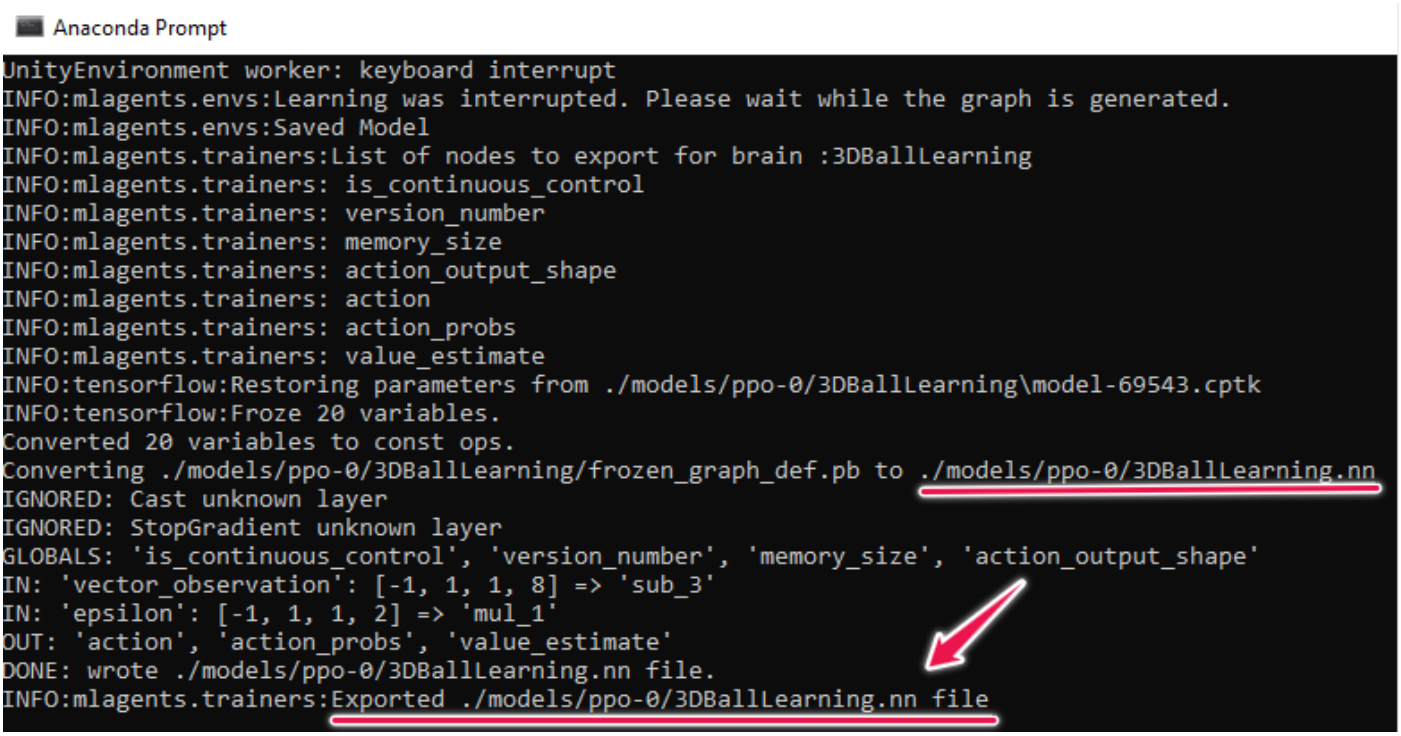

Figura 4.7. Salida de la consola de comandos de Anaconda al terminar el entrenamiento

4.2.- Uso de la librería ml-agents para selección de ruta óptima

Ya sabemos cómo funciona la librería de Unity ml-agents y cómo entrenar a los agentes. Pero nuestro objetivo es estudiar algoritmos de machine learning de selección de rutas óptimas. Por desgracia, la librería ml-agents no cuenta con un algoritmo que realice dicha función entre sus muchos ejemplos que trae.

Para poder seleccionar una ruta óptima debe haber rutas posibles. Eso conlleva a que es necesario que haya un punto de partida y un punto de llegada. Es decir, el agente pretenderá ir de un punto a otro dentro de un escenario.

Siguiendo este principio básico, vamos a ver cómo entrenar a un agente para que sepa llegar a un punto en concreto. Como primer experimento, en la

escena únicamente estarán el agente y el punto de destino, sin ningún tipo de obstáculo. Así, nuestro primer escenario quedaría de la siguiente manera:

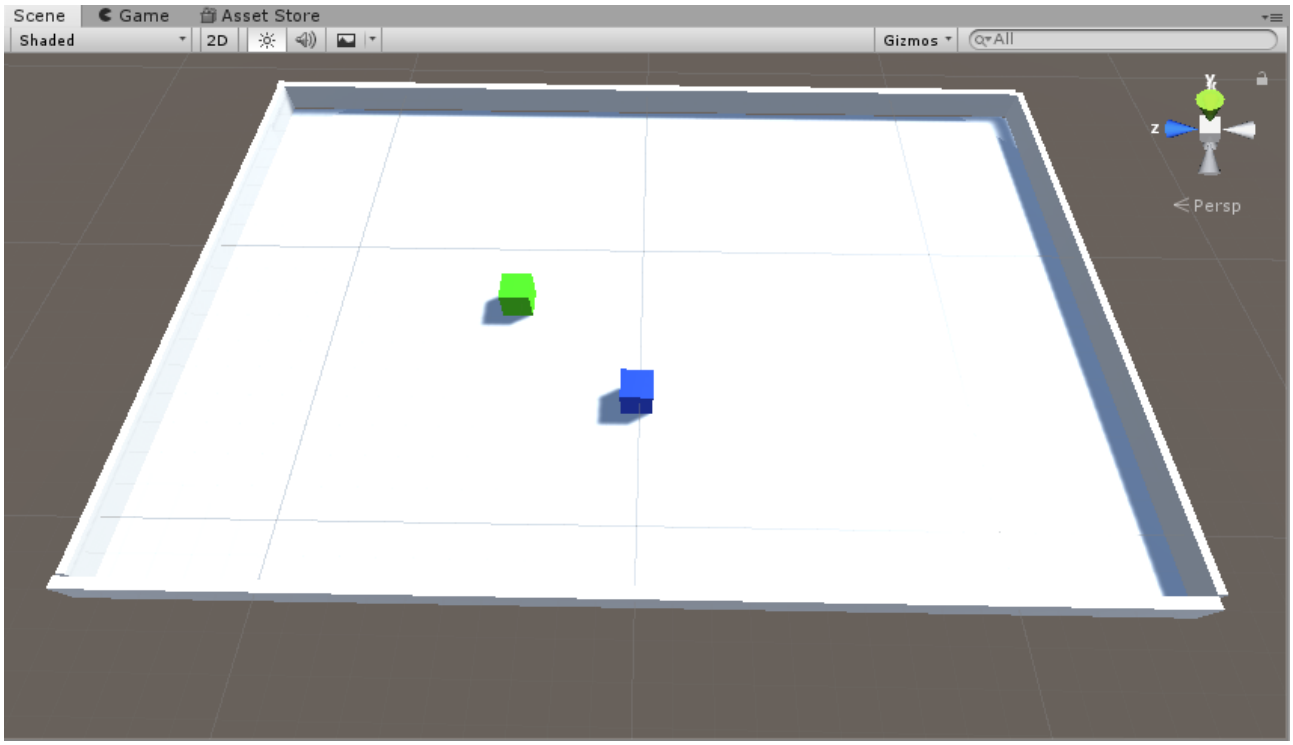

Figura 4.8. Escenario del primer experimento

El cubo azul representa al agente y el cubo verde el punto de destino que debe alcanzar dicho agente.

A continuación, vamos a ver el script del agente. Dicho script, como ya vimos anteriormente, debe extender a la clase *Agent* de la API de la librería de ml-agents.
Al comienzo de cada una de las iteraciones, se establece la posición del agente en el punto (0,0,0), mediante la función *AgentReset()*.
Las entradas de la red neuronal se recogen mediante la función *CollectObservations()*. Para cada una de las entradas se utiliza la función *AddVectorObs()*, donde se indica qué es lo que va a tener en cuenta el cerebro sobre el entorno, para poder luego tomar las decisiones.
En nuestro caso, como valores de entrada, nos interesa la posición del agente en el eje X y en el eje Z y la posición del punto destino, también en el eje X y en el eje Z. Así, el cerebro, a la hora de tomar las decisiones, tendrá en cuenta en qué posición se encuentran tanto el agente como la meta.
La decisión tomada se realiza mediante la función *AgentAction()* que se ejecuta cada frame. En nuestro experimento, en cada frame que transcurre, el agente recibe un castigo pequeño, de -0.1, de manera que cuanto más tiempo esté el agente buscando el objetivo más castigo conllevará, así tratará de encontrar el punto de destino en menor tiempo. Esto no tiene por qué ser así, ya que en un ambiente urbano el camino óptimo no será el camino más corto

debido a qué el tráfico, entre otros, también influye. Pero en nuestro primer ejemplo, al ser tan sencillo puede valer.
El vector de decisión se compone de dos valores: uno para el movimiento en el eje X y otro para el movimiento en el eje Z. El agente se moverá en una dirección u otra dependiendo de estos valores.
Una vez que el agente se ha movido obedeciendo a la decisión del cerebro, se pasa a compensar o castigar dependiendo de qué haya ocurrido. En este ejemplo tan simple, se tiene en cuenta si la posición final del agente con respecto al punto destino después de llevar a cabo el movimiento es menor que la posición antes de realizarlo. Si luego de moverse, se encuentra más cerca del objetivo, se le recompensa, en caso contrario se le castiga.
Cuando el agente alcanza el punto objetivo se le debe recompensar. Para ello, Unity cuenta con una función *OnTriggerEnter()* que detecta cuándo un objeto entra en el área de colisión de otro objeto. En este caso, el área de colisión del cubo objetivo es del mismo tamaño que el propio cubo, por lo que el agente entrará en dicho área en cuanto toque al cubo objetivo. Esto significa que la función se ejecutará en cuando el agente toque el cubo destino. Estas áreas de colisión debe tener un “tag” asociado para poder distinguir en el código en qué área o áreas se ha entrado. El tag del destino es *goal*. Al ejecutarse esta función, como el agente ha conseguido su objetivo que era llegar al punto de destino, se le recompensa con una puntuación positiva de 30. Además, dentro de esta misma función, se ejecuta otra función *ChangeGoalPosition()* creada por nosotros que cambia de posición el destino del agente, de manera que no siempre esté en la misma posición, ya que esto haría que, una vez entrenado el modelo, si el destino es cambiado de posición, el agente no sabría llegar a él, en lugar de eso, seguiría yendo a la posición donde se encontraba el destino cuando se entrenó.
Por último, se ejecuta la función *Done()* que se encarga de finalizar la iteración, resetear el agente y comenzar una nueva iteración.

```
public override void AgentReset()
{
    gameObject.transform.localPosition = Vector3.zero;
}

23 referencias
public override void CollectObservations()
{
    AddVectorObs(transform.localPosition.x);
    AddVectorObs(transform.localPosition.z);
    AddVectorObs(goal.transform.localPosition.x);
    AddVectorObs(goal.transform.localPosition.z);
}

22 referencias
public override void AgentAction(float[] vectorAction, string textAction)
{
    AddReward(-0.1f);
    float initialDistance = Vector3.Distance(gameObject.transform.localPosition, goal.transform.localPosition);

    float newX = transform.localPosition.x + (vectorAction[0] * speed * Time.deltaTime);
    float newZ = transform.localPosition.z + (vectorAction[1] * speed * Time.deltaTime);
    transform.localPosition = new Vector3(newX, transform.localPosition.y, newZ);

    float finalDistance = Vector3.Distance(gameObject.transform.localPosition, goal.transform.localPosition);

    if(initialDistance < finalDistance)
    {
        AddReward(-0.5f);
    }
    else
    {
        AddReward(0.2f);
    }

}
```

Figura 4.9. Script del experimento (1)

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("goal"))
    {
        AddReward(30f);
        ChangeGoalPosition();
        Done();
    }
}

1 referencia
private void ChangeGoalPosition()
{
    goal.transform.localPosition = new Vector3(Random.Range(-1f, 13f), goal.transform.localPosition.y, Random.Range(-11f, 11f));
}
```

Figura 4.10. Script del experimento (2)

Ya tenemos un primer paso inicial: hemos conseguido que el agente aprenda a llegar al punto de destino. Sin embargo, este experimento ha sido bastante sencillo. En una vía urbana, el agente debe seguir un camino concreto que está marcado por la carretera, es decir, no tiene movimiento libre por todo el escenario como en el experimento anterior.

Para simular la carretera añadiremos paredes que formen un recorrido. Así, el nuevo escenario quedará de la siguiente manera:

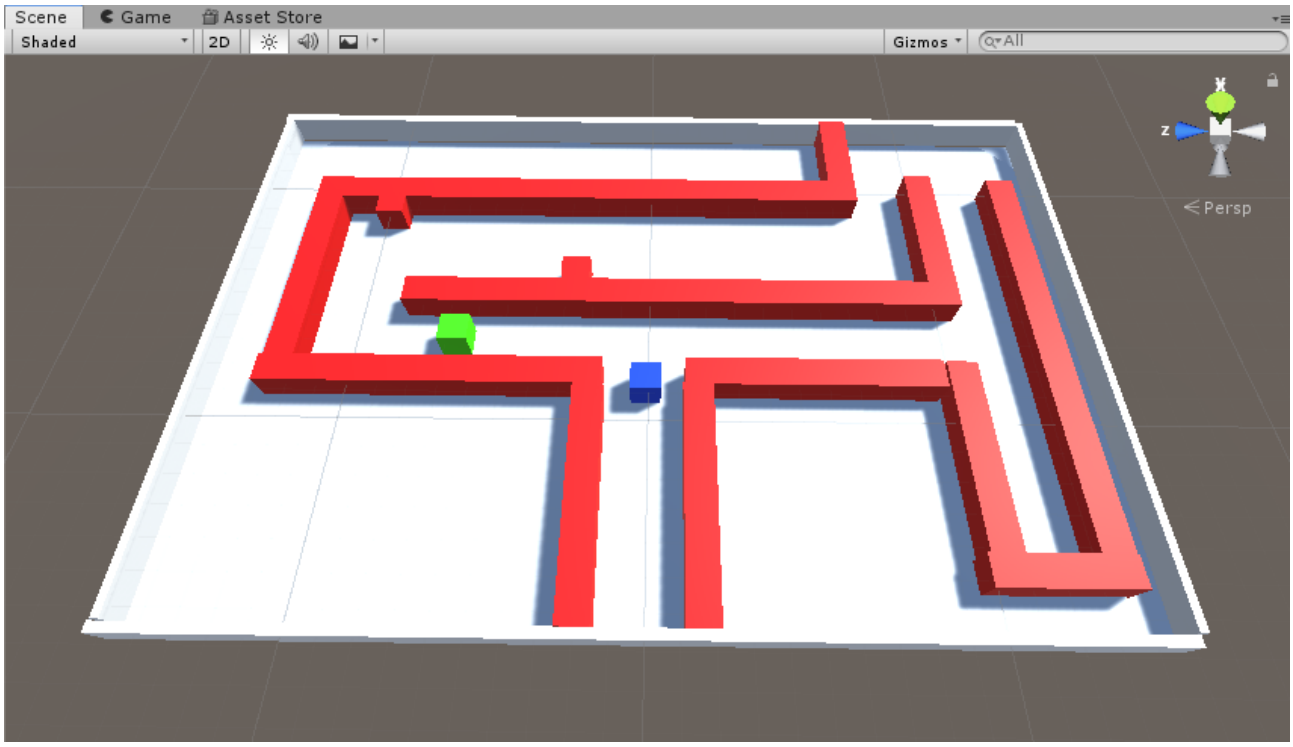

Figura 4.11. Escenario del segundo experimento

Así, el agente deberá seguir el camino marcado por estas paredes, como si fuera la carretera. Ahora la red neuronal no debe tener en cuenta solamente la posición del agente y la del punto objetivo, sino que también deberá tener en cuenta la posición de estas paredes.

```
public override void CollectObservations()
{
    AddVectorObs(transform.localPosition.x);
    AddVectorObs(transform.localPosition.z);
    AddVectorObs(goal.transform.localPosition.x);
    AddVectorObs(goal.transform.localPosition.z);

    AddVectorObs(obstacles.transform.localPosition.x);
    AddVectorObs(obstacles.transform.localPosition.z);
    AddVectorObs(walls.transform.localPosition.x);
    AddVectorObs(walls.transform.localPosition.z);

    AddVectorObs(obstacles);
    AddVectorObs(walls);
}
```

Figura 4.12. Nuevas entradas para la red neuronal

Además de añadir la posición de dichas paredes, también se ha añadido el objeto en sí, para estar totalmente seguros de que la red neuronal las tendrá en cuenta. Hay otro segundo objeto, que son las cuatro paredes que delimitan al escenario.

El último cambio con respecto al script del primer experimento es la nueva condición dentro de la función que detecta una colisión.

Figura 4.13. Nueva colisión

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("goal"))
    {
        AddReward(30f);
        ChangeGoalPosition();
        Done();
    }
    else if (other.CompareTag("Obstacle") || other.CompareTag("wall"))
    {
        AddReward(-10f);
        Done();
    }
}
```

En el experimento anterior, solamente teníamos la colisión con el punto objetivo, con *tag* “goal”. En cambio ahora debemos añadir la nueva colisión tanto con las paredes que forman la carretera como las que delimitan el escenario. Estas paredes tienen los *tags* “Obstacle” y “wall” respectivamente.
Cuando el agente choque con estas paredes, recibirá un castigo de -10, para que aprenda que no debe chocar con las mismas. De esta manera, al evitarlas, seguirá el camino que forma la carretera.

Sin embargo, el objetivo se ha complicado con respecto al experimento anterior, y este script tan sencillo, a pesar de que en un principio puede dar a pensar que funciona, no es lo suficientemente completo para conseguir el objetivo que queremos. Tras muchas iteraciones en el entrenamiento (más de 200.000), el agente ha aprendido a llegar al objetivo y a esquivar las paredes, pero aún así no llega a esquivarlas del todo, se atasca, sobre todo en las esquinas donde tiene que girar (figura 4.14).

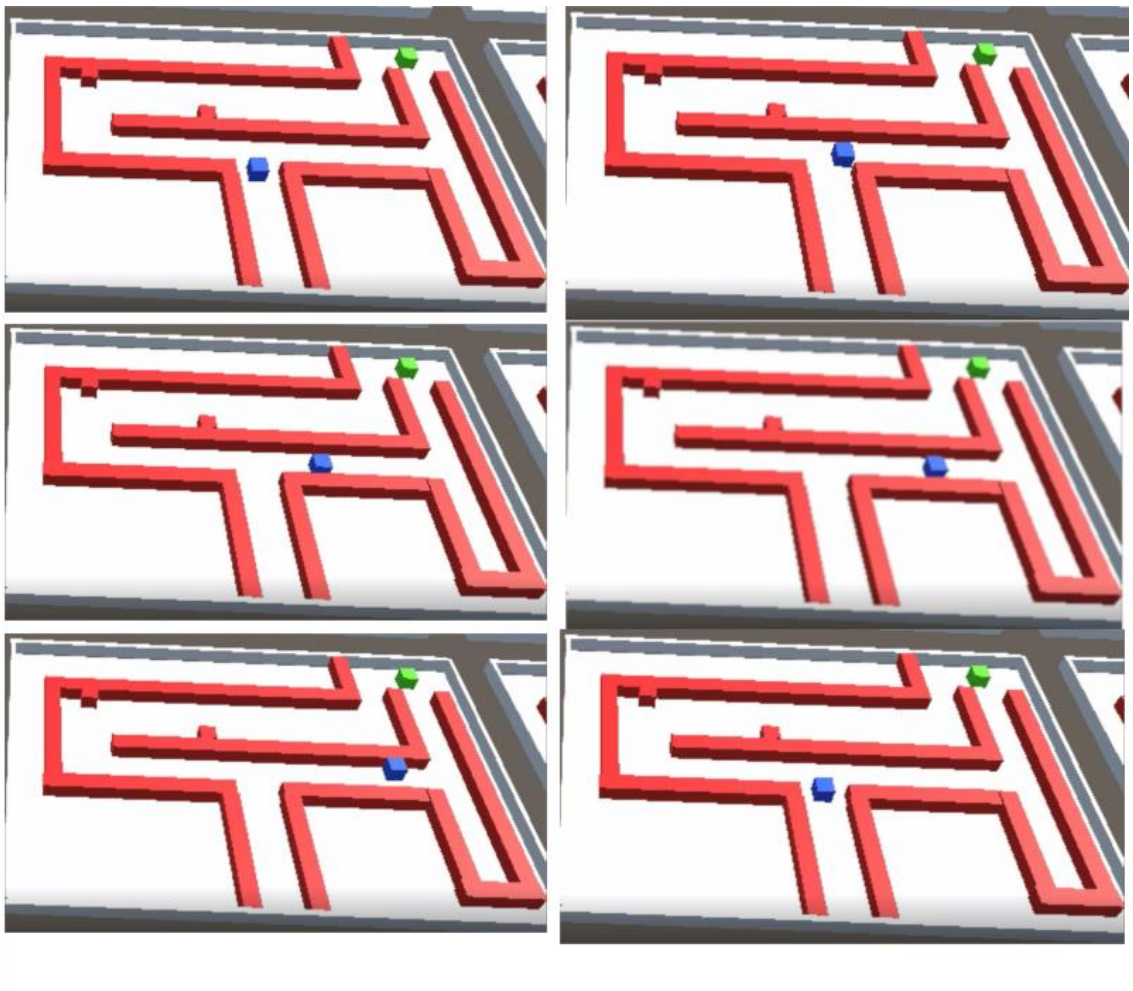

Figura 4.14. Capturas temporales del entrenamiento (de izquierda a derecha y de arriba a abajo)

Se observa claramente que aprende a ir hacia el punto destino y también que no debe tocar las paredes, por eso sale de la “calle” inferior, gira a la derecha y sigue por esa “calle”. Pero cuando debe volver a girar hacia arriba para ir al objetivo, no termina de aprender a esquivar la esquina y vuelve a su posición inicial en una nueva iteración. Por lo tanto, descartamos este experimento.

Para el siguiente experimento, en lugar de usar un terreno amplio y delimitar los caminos con paredes, usaremos como escenario prefabs de carreteras (figura 4.15), de manera que si el agente se sale de una de las carreteras, caerá. Cuando caiga, debe recibir un castigo, para así aprender a permanecer en la carretera, al mismo tiempo que también debería aprender a ir al objetivo.

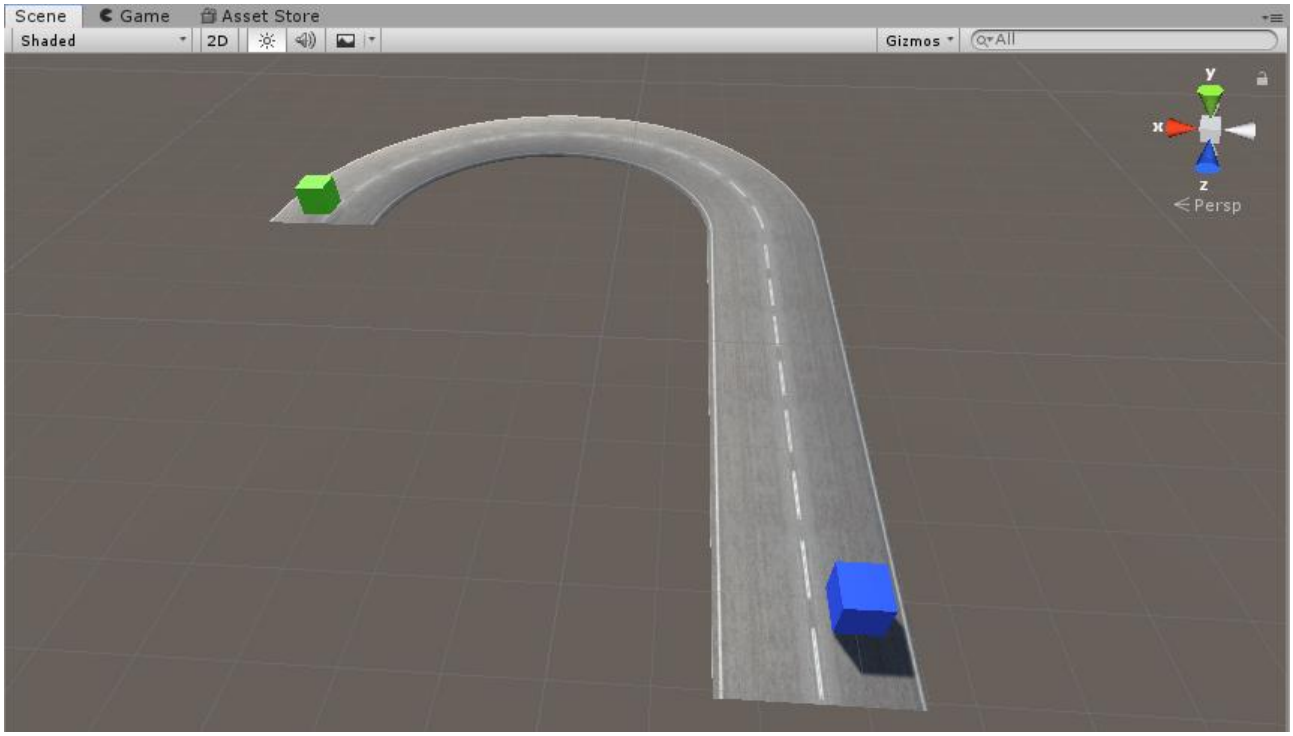

Figura 4.15. Escenario del tercer experimento

Nuevamente, partimos del script del primer experimento. Como en este caso no tenemos las paredes que delimitaban el camino o carretera, éstas ya no serán entradas de la red neuronal, por lo tanto, ya no deben recogerse como observaciones del entorno en el método *CollectObservations()*. En su lugar, lo que sí es necesario que el algoritmo tome en cuenta es la carretera en sí. Por ello, sustituimos dichos inputs de la red neuronal (las paredes) por el objeto de Unity que es la carretera.

```
public override void CollectObservations()
{
    AddVectorObs(transform.localPosition.x);
    AddVectorObs(transform.localPosition.z);
    AddVectorObs(goal.transform.localPosition.x);
    AddVectorObs(goal.transform.localPosition.z);

    AddVectorObs(road);

    //AddVectorObs(obstacles.transform.localPosition.x);
    //AddVectorObs(obstacles.transform.localPosition.z);
    //AddVectorObs(walls.transform.localPosition.x);
    //AddVectorObs(walls.transform.localPosition.z);

    //AddVectorObs(obstacles);
    //AddVectorObs(walls);
}
```

Figura 4.16. Nuevas entradas de a red neuronal

Como se puede observar en a figura 4.16, se han suprimido los método *AddVectorObs()* que recogían las paredes del experimento anterior y se ha añadido uno nuevo que recoge la carretera (road) por la que pasará el agente.

Como se ha dicho anteriormente, el agente será castigado cuando caiga de la carretera. Para ello, es necesario modificar el método *AgentAction()* en el que se lleva a cabo la decisión y se evalúan las consecuencias de las mismas. Dicha modificación consiste en añadir un nuevo condicional que compruebe la posición del agente. Esto es, comprueba si el agente ha caído de la carretera.

```
public override void AgentAction(float[] vectorAction, string textAction)
{
    AddReward(-0.1f);
    float initialDistance = Vector3.Distance(gameObject.transform.localPosition, goal.transform.localPosition);

    float newX = transform.localPosition.x + (vectorAction[0] * speed * Time.deltaTime);
    float newZ = transform.localPosition.z + (vectorAction[1] * speed * Time.deltaTime);
    transform.localPosition = new Vector3(newX, transform.localPosition.y, newZ);

    float finalDistance = Vector3.Distance(gameObject.transform.localPosition, goal.transform.localPosition);

    if (initialDistance < finalDistance)
    {
        AddReward(-0.5f);
    }
    else
    {
        AddReward(0.2f);
    }

    if (gameObject.transform.localPosition.y < 2.6f)
    {
        AddReward(-50f);
        Done();
    }

}
```

Figura 4.17. Nuevo condicional para la recompensa/castigo

Si el agente ha caído, quiere decir que se ha salido de la carretera. Para ello, se comprueba la posición del agente en el eje Y (eje vertical) para determinar si se ha caído de la carretera o no. Si, efectivamente, el agente ha caído, es decir, se ha salido de la carretera, recibirá un castigo de -50 y volverá a repetirse una nueva iteración.
Estas son las únicas modificaciones con respecto al script del experimento anterior.

Ahora bien, ¿será el agente capaz de llegar al destino sin salirse de la carretera? Tras muchas iteraciones en el aprendizaje (unos 250.000 pasos) con 12 escenarios aprendiendo simultáneamente (para que el aprendizaje sea mucho más rápido, 12 veces más rápido concretamente), efectivamente el agente aprende a llegar al objetivo sin caerse (salirse) de la carretera, tal y como muestra la siguiente figura:

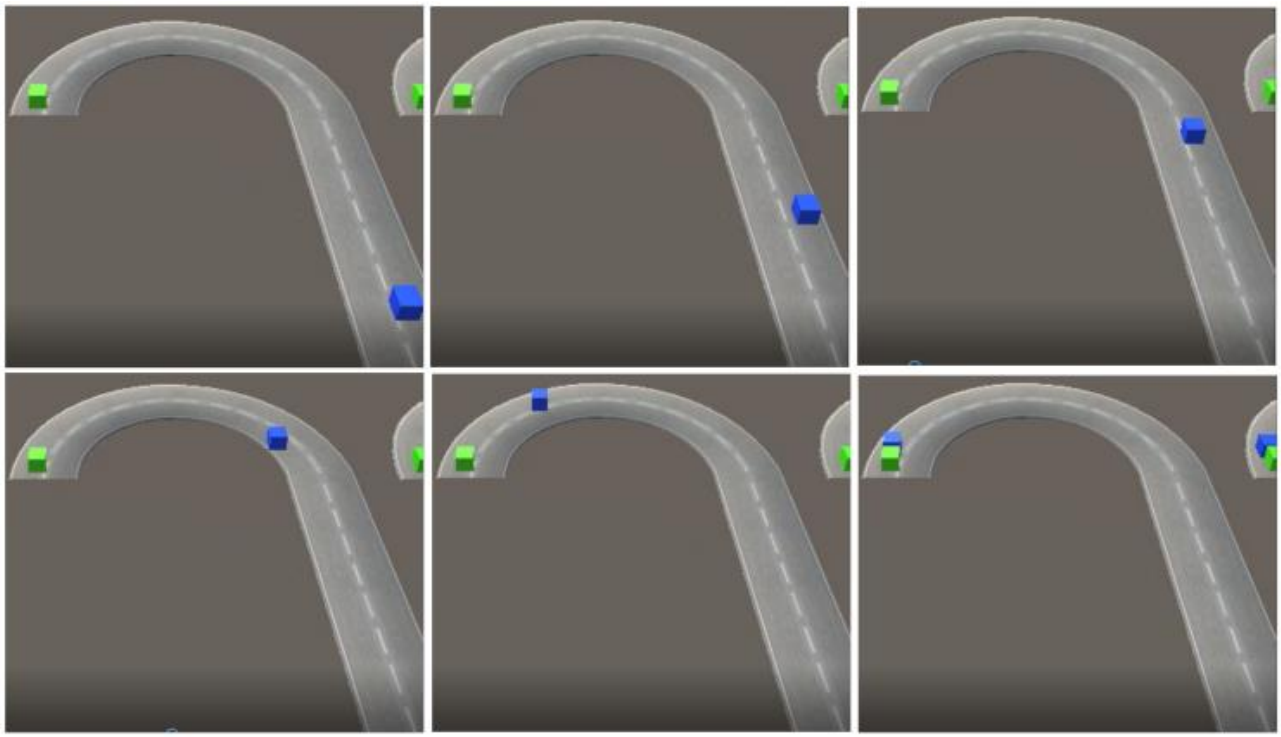

Figura 4.18. El agente aprende en el segundo experimento

Hemos conseguido que, mediante su propio aprendizaje, el agente sea capaz de llegar al objetivo siguiendo la carretera. Sin embargo, en este ejemplo solamente hay un único camino. Nos interesa saber si el agente es capaz de, dentro de un escenario con más de un camino, alcanzar el objetivo por el camino más corto que es, a priori y sin tener en cuenta el tráfico y atascos, el más óptimo.

Para ello, en lugar de ser una único camino, crearemos un nuevo entorno en el que el agente pueda llegar al destino por varios caminos diferentes, tal y como se muestra en la figura 4.19.

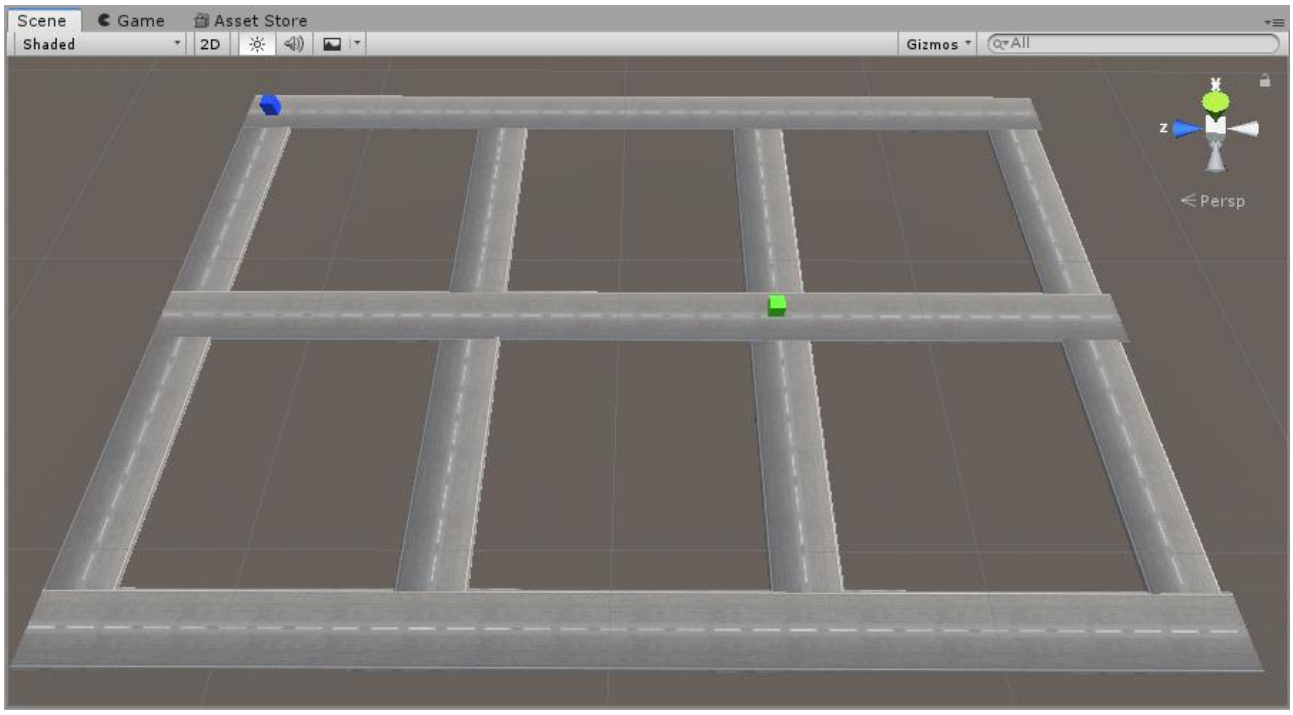

Figura 4.19. Escenario con varios caminos

Sin embargo, después de muchos entrenamientos se ha observado que el agente no sólo no elige el camino más corto, sino que además no termina de llegar al punto objetivo. No obstante, tras un largo entrenamiento (puede que hasta días) el agente aprendería a llegar a su objetivo, pero esto no es nada viable.

4.3.- Conclusiones del uso de la librería ml-agents

En conclusión, la librería ml-agents es adecuada para cualquier tipo de algoritmo de machine learning. Sin embargo, es muy posible que sea necesario que el propio usuario tenga que definir dicho algoritmo, ya que, dado el amplio abanico de posibilidades que ofrece ml-agents, solamente existen unos pocos ejemplos y algoritmos ya definidos. Para el algoritmo de nuestro proyecto no existe ya ninguno creado para ml-agents, aunque es totalmente viable la definición de un algoritmo de este tipo.

No obstante, nuestro proyecto no consiste en definir ningún algoritmo, sino probar la viabilidad de utilizar la herramienta Unity con este fin. Efectivamente, al menos por medio de la librería de ml-agents, Unity es totalmente viable de ser utilizado para la definición y posterior aplicación de algoritmos de machine learning de selección de ruta óptima dentro de una trama urbana.

De hecho, en la página oficial de Unity hay algunos trabajos hechos con la librería ml-agents. Cada uno de ellos es un algoritmo totalmente diferente al

resto, por lo que se puede comprobar que es posible definir cualquier tipo de algoritmo de machine learning con dicha librería. Estos trabajos se pueden ver en **https://connect.unity.com/challenges/ml-agents-1**.

5.- INTELIGENCIA ARTIFICIAL *NAVIGATION AND PATHFINDING*

Unity cuenta con una inteligencia artificial ya integrada que consiste en el cálculo de la ruta más corta entre el agente y un objetivo. Esta es la inteligencia artificial que se utiliza en videojuegos para que un enemigo persiga al jugador, entre otras cosas.

El sistema de navegación de Unity permite a un agente moverse de manera inteligente sobre el terreno, utilizando mallas que se crean automáticamente a partir de la geometría de dicho terreno.
Según la documentación de Uniy, “le proporciona al personaje la habilidad de entender que necesita tomar unas escaleras para llegar al segundo piso, o saltar para pasar sobre una zanja”. Es decir, el agente sabe por dónde puede o deber ir para alcanzar el objetivo. Podemos utilizar esto a nuestro favor para dirigir el coche a través de la carretera.

5.1.- Partes del sistema de navegación.

El sistema de navegación de Unity está formado por las siguientes piezas:

- **NavMesh**: Es una estructura de datos que define las superficies por las que el agente dentro del escenario y permite encontrar un camino de una superficie a otra. La estructura se construye automáticamente a partir de la geometría del escenario.

- **NavMesh Agent**: Componente que permite crear personajes u objetos que deben llegar a un objetivo. Cada agente observa el escenario usando la malla *NavMesh* y saben cómo llegar al objetivo, al mismo tiempo que evitan obstáculos y otros agentes.

- **Off-Mesh Link**: Es un componente que permite realizar atajos por lugares por los que el agente no puede pasar, como por ejemplo saltando una pared.

- **NavMesh Obstacle**: Son obstáculos que los agentes deberán esquivar mientras se mueven por el escenario. Además, si el obstáculo

bloquea el camino que el agente debería seguir, éste puede encontrar otro camino diferente.
Para nuestro proyecto sería interesante utilizar este componente para simular tráfico o para impedir que un coche vaya en dirección contraria en la carretera.

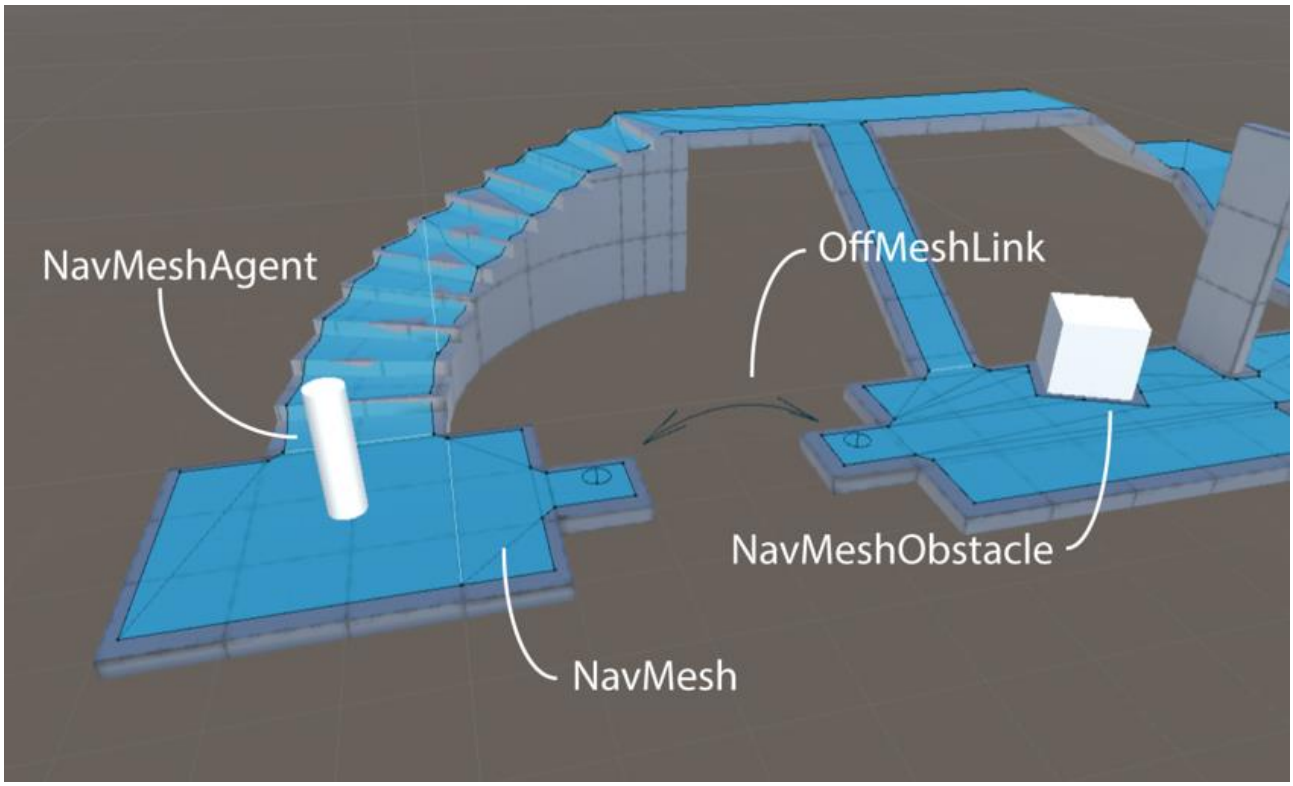

Figura 5.1. Partes que componen el sistema de navegación

5.2.- Cómo funciona el sistema de navegación.

Antes de empezar a utilizar este sistema de inteligencia artificial, veremos cómo funciona.
Para empezar, es necesario que el agente sepa por qué superficies de la escena puede pasar y por cuáles no. Las áreas por las que el agente puede pasar se conoce como NavMesh (Navigation Mesh) y se calcula a partir de la geometría del escenario automáticamente. El NavMesh almacenan la superficie como polígonos convexos, tal y como se puede apreciar en la figura 5.2.

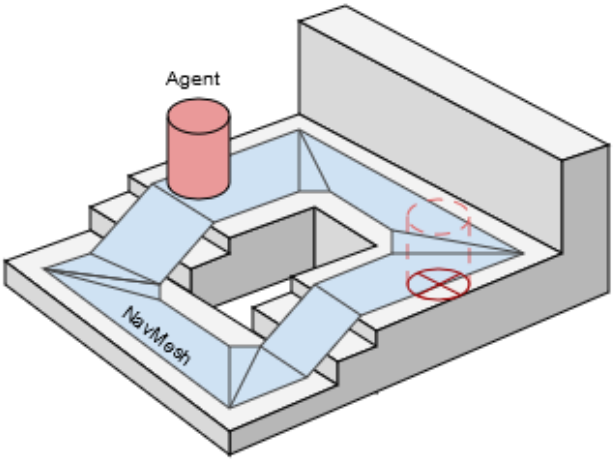

Figura 5.2 Representación del NavMesh

Una vez que ya tenemos las áreas por el que el agente puede pasar, debemos encontrar un camino hasta el objetivo.
Para encontrar un camino entre dos localizaciones en el NavMesh, primero debemos asociar tanto el origen como el destino con el polígono más cercano a ellos en la malla. Teniendo ya el polígono de inicio y el polígono de destino, buscamos los polígonos que conducen hasta el destino.
El algoritmo que utiliza Unity para realizar este proceso es A*.

5.2.1. - Algoritmo A*

Es un algoritmo de búsqueda para el cálculo de caminos en una red, en nuestro caso dentro de la malla que forma la superficie por el que el agente puede pasar. Es un algoritmo heurístico, ya que utiliza una función de evaluación por la cual se evalúan los diferentes nodos de la red y servirá para calcular la probabilidad de que dichos nodos pertenezcan al camino óptimo. Dicha función es, para cada nodo n:

f(n) = g(n) + h(n)

donde g(n) indica la distancia desde el nodo de origen hasta dicho nodo n, es decir, el coste del camino recorrido hasta él; y h(n) indica la distancia estimada desde el nodo n a evaluar hasta el nodo de destino.

La función h(n) es una función heurística que expresa cómo de lejos se está aún de llegar al nodo destino. Esta función no debe sobrestimar el coste real de alcanzar el nodo destino, por lo tanto, h(n) deberá ser menor que h*(n). De esta manera, la función h(n) es una función heurística admisible, y garantiza el hallazgo de la solución óptima para el problema del camino más corto.

En la figura 5.3 se puede observar un ejemplo de aplicación del algoritmo A* en un escenario cuadriculado para encontrar el camino más corto entre la casilla de inicio pintada en verde y la casilla destino pintada en azul. Cada casilla posee un número que es el valor del coste de desplazarse desde la casilla inicial hasta dicha casilla, tal y como indica la función g(n).

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Figura 5.3. Ejemplo del algoritmo A*

Así, se ha conseguido encontrar el camino más corto desde la casilla de inicio hasta la casilla destino. Este será el algoritmo que aplicará Unity en nuestra malla para encontrar el camino más corto que deberá seguir nuestro coche para llegar al destino.

A la hora de moverse el agente por la malla del escenario, siempre avanzará hacia la esquina más cercana, tal como se muestra en la figura 5.4. De esta manera, el coche no tomará las curvas en la carretera, si no que irá en línea recta hacia el borde de la carretera, de la misma manera que se muestra en la figura 5.4.

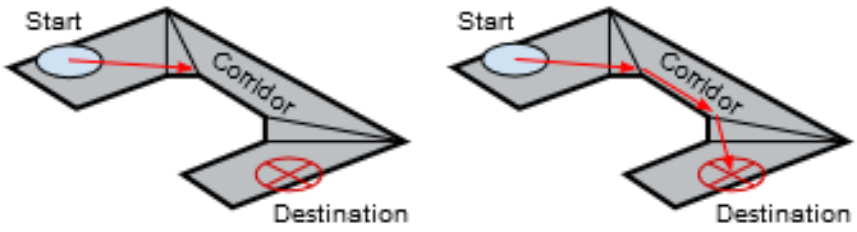

Figura 5.4. Recorrido que sigue el agente en una curva o esquina

Por lo tanto, para que la conducción sea apropiada, deberemos tener esto en cuenta.

Si aplicamos el cálculo del NavMesh a nuestra carretera, toda la superficie será suelo por el que el agente pueda pasar, de manera que, una vez aplicada la malla, se verá de la siguiente manera:

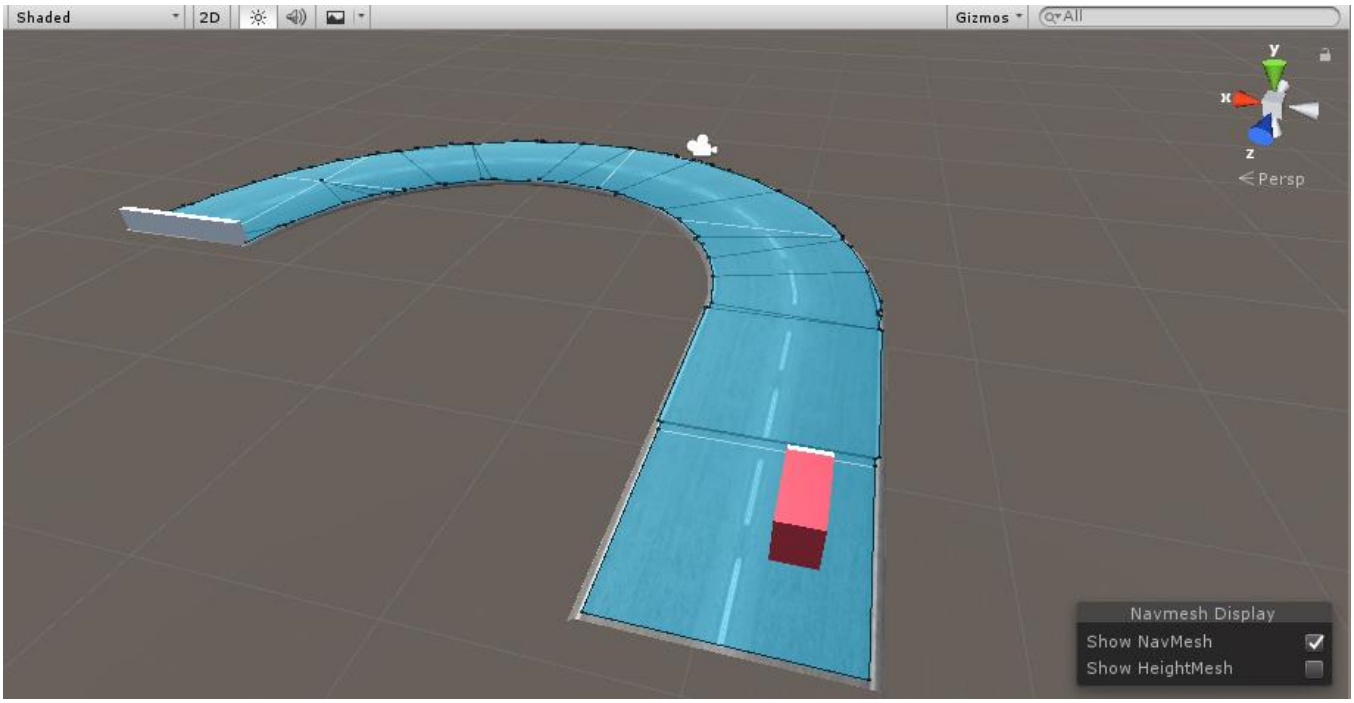

Figura 5.5. NavMesh aplicada a la carretera

Así, como hemos comentado anteriormente, el coche (cubo rojo) a la hora de alcanzar su objetivo (pared blanca) no seguirá el carril, sino que, siguiendo el camino más corto, irá directamente hacia la curva, tal y como se puede observar en la figura 5.6.

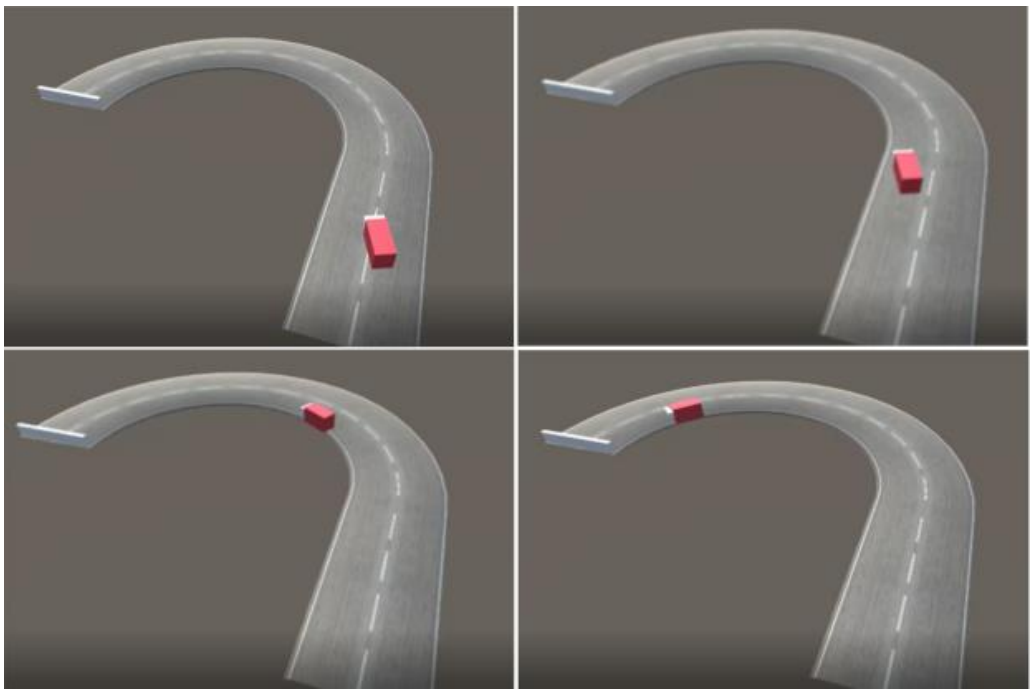

Figura 5.6. El agente toma la curva por el camino más corto sin seguir el carril

Una manera de evitar que esto suceda es limitar la malla únicamente a los carriles, de esta manera el coche no se saldrá de ellos. Para conseguirlo, una buena manera es añadiendo “paredes” que limiten los carriles y que no se tengan en cuenta a la hora de calcular la malla.
De este modo, el prefab de la carretera quedará de la siguiente manera:

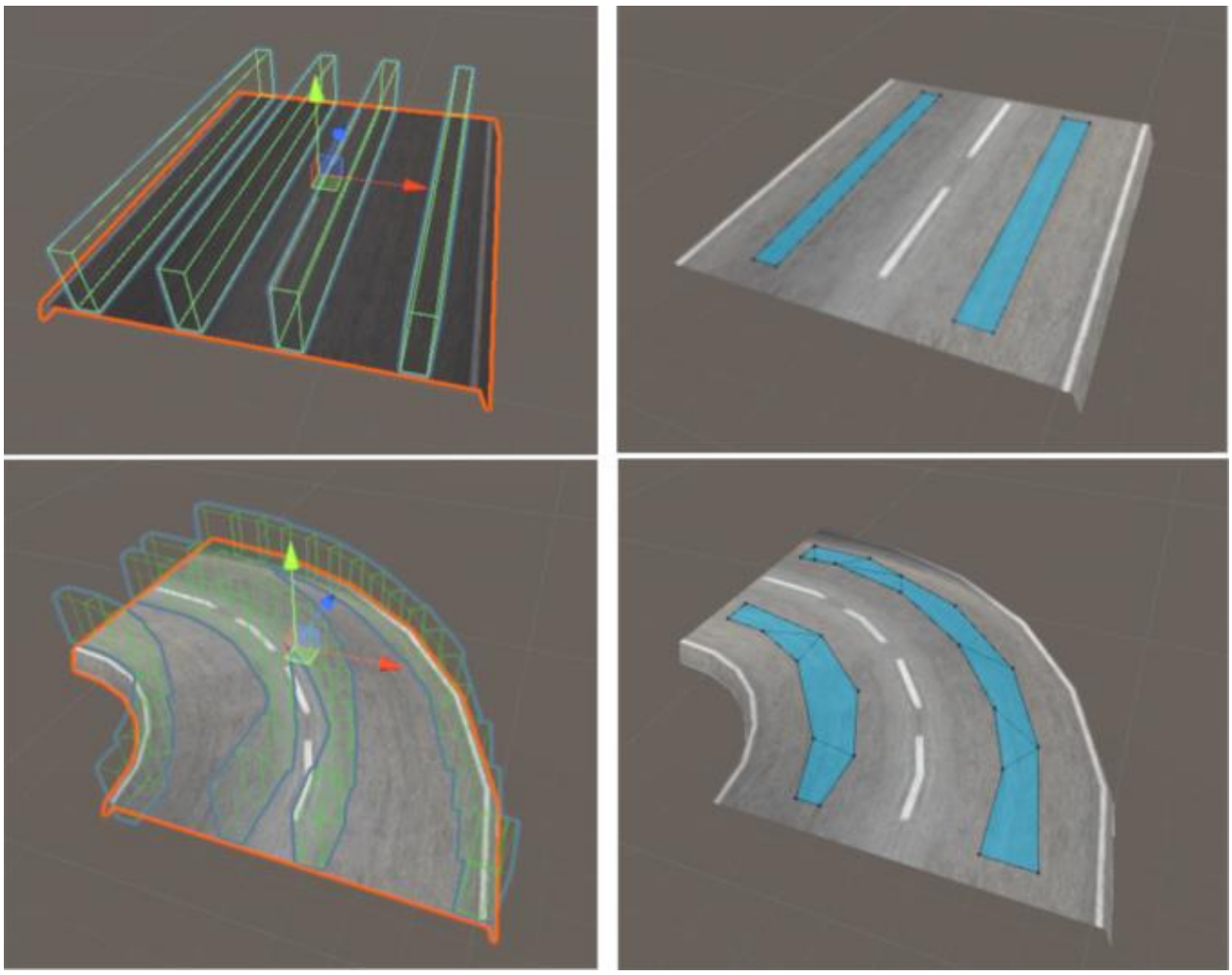

Figura 5.7. Prefabs de las carreteras y sus correspondientes mallas

De esta manera, se puede observar que se ha solucionado este problema, y ahora el coche solamente podrá andar por su carril correspondiente.

5.3.- Simulaciones del sistema de navegación

Ya está listo para probar si esta inteligencia artificial es válida para nuestro propósito. Para ello, crearemos una carretera en un entorno urbano y colocaremos un objetivo al que el agente deberá llegar para observar si éste elige el camino más corto o no.

Figura 5.8. Entorno donde probaremos la IA de navegación

Nuestro coche está representado por el cubo rojo, mientras que el destino será el cilindro blanco del fondo de la imagen.
Antes de simularlo, indicaremos qué camino deberá escoger el agente para llegar al cilindro. En este caso, cuando llegue a la bifurcación del camino, deberá coger el camino de la derecha, tal y como se muestra en la figura 5.9.

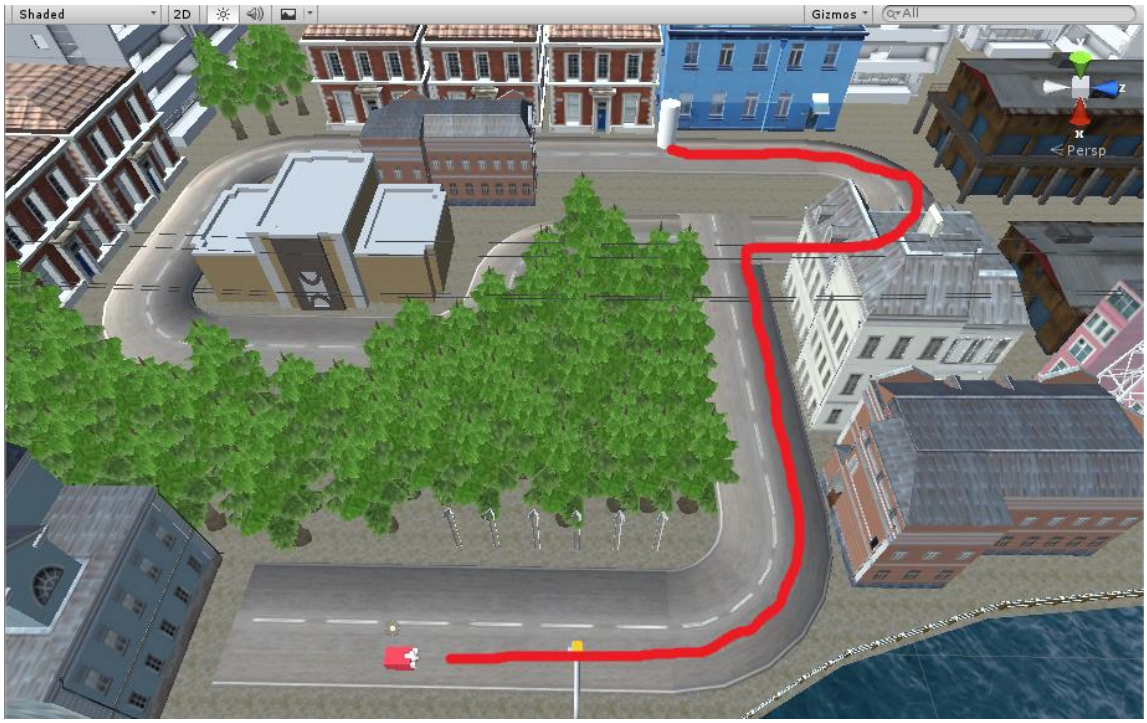

Figura 5.9. Recorrido que deberá realizar el agente para llegar al destino

Ahora sí, probaremos la viabilidad de este algoritmo. Como se puede observar en la figura 5.10 (de izquierda a derecha y de arriba a abajo), efectivamente el agente ha cogido el camino esperado para llegar al objetivo. En cada una de las cuatro imágenes, la parte de la izquierda muestra lo que ve el coche mientras que la parte de la derecha muestra una vista general del escenario.

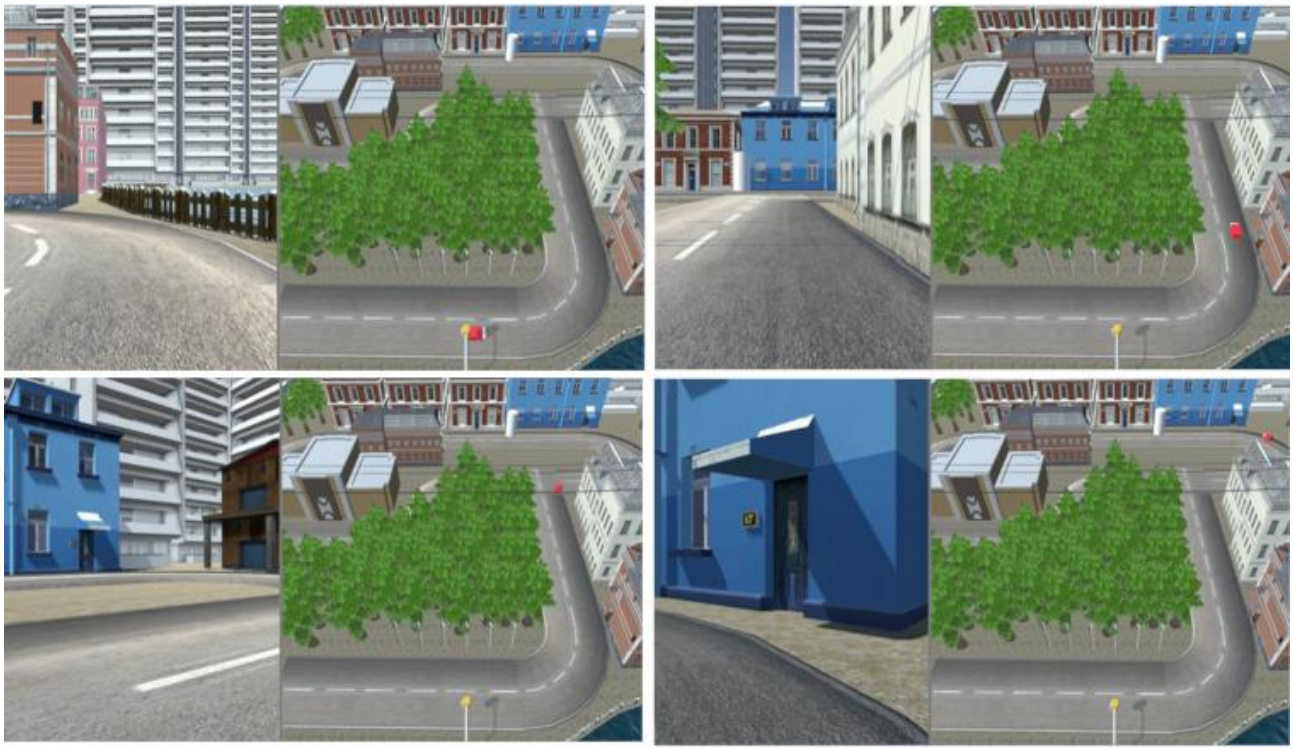

Figura 5.10. El agente escoge el camino más corto para llegar al destino

Sin embargo, podría haber sido casualidad que el coche eligiese el camino más corto hasta el destino, por lo que, para estar totalmente seguro, cambiaremos el destino de lugar, de manera que ahora el camino más corto sea el mostrado en la figura 5.11.

Figura 5.11. Camino que deberá recorrer el coche en esta segunda prueba

Como se puede observar, ahora el destino (indicado como un cilindro blanco) se encuentra más cercano siguiendo el camino de la izquierda de la bifurcación, por lo que, si este algoritmo de inteligencia artificial es eficaz, el coche deberá elegir dicho camino.

Una vez más, tras realizar la simulación en Unity, el coche efectivamente ha elegido el camino más corto para llegar al destino, tal y como se puede observar en la figura 5.12.

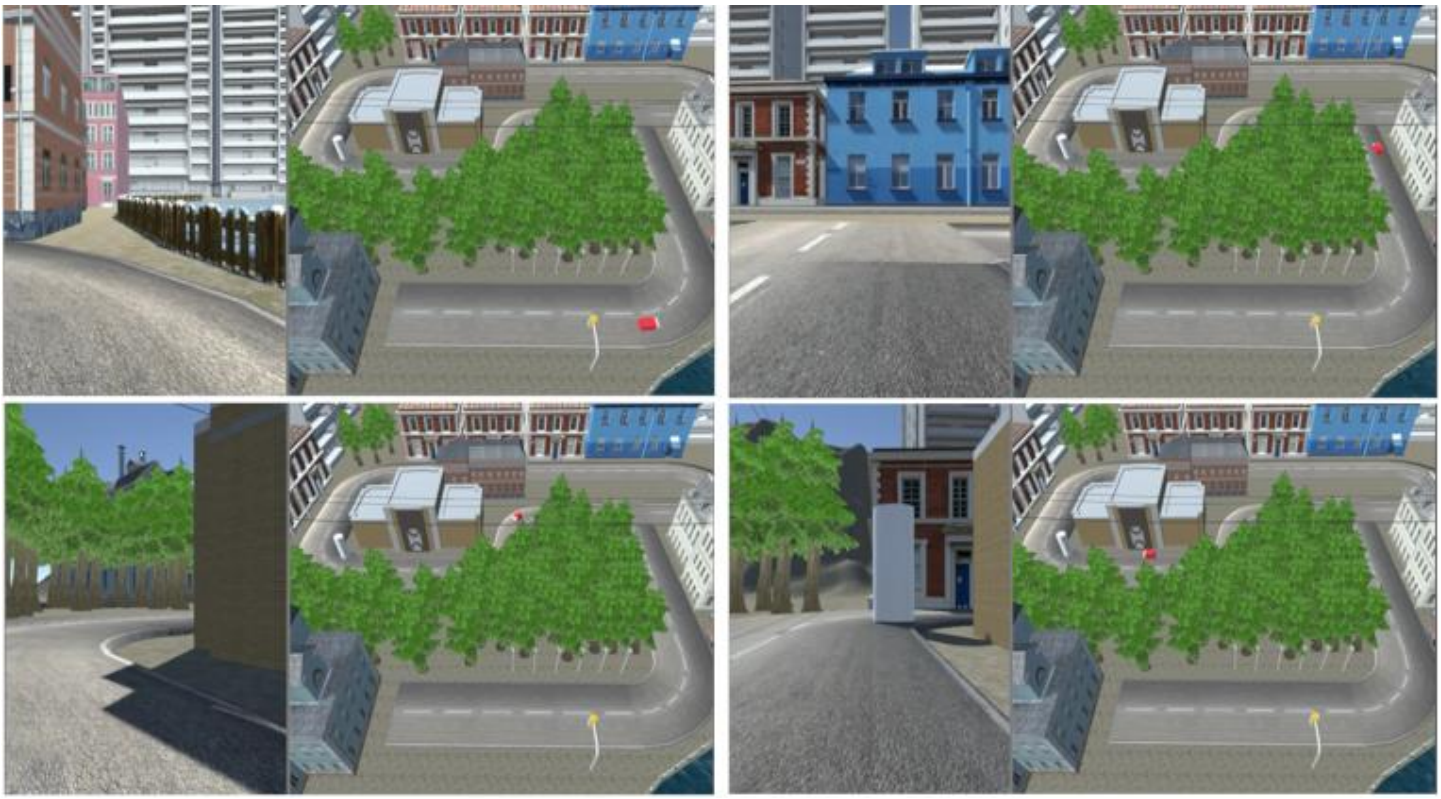

Figura 5.12. El coche vuelve a elegir el camino más corto hasta el destino

Así, podemos decir que el algoritmo de inteligencia artificial de navegación de Unity sí es viable para nuestro caso. Sin embargo, tiene algunas desventajas como por ejemplo las modificaciones que han sido necesarias realizar como señalizar mediante paredes los carriles dentro de las carreteras. Otro inconveniente es que, en el supuesto de que el camino más corto hasta el destino fuese dar media vuelta, es decir, ir en sentido contrario en el carril, el coche irá en sentido contrario. Este inconveniente se podría solucionar mediante obstáculos, de manera que justo detrás del coche hubiese un obstáculo que le impidiese escoger ese camino.

5.4.- Conclusiones del uso del sistema de navegación

El algoritmo de inteligencia artificial del sistema de navegación es perfectamente viable para nuestra finalidad, ya que está pensado para elegir el camino más corto entre dos puntos. Sin embargo, es necesario hacer algunas modificaciones dentro del escenario para que este algoritmo pueda llegar a funcionar correctamente. Además, este algoritmo no podía ser aplicado, por ejemplo, a un algoritmo de simulación de conducción, ya que el propio algoritmo de navegación ya mueve al agente. Dicho algoritmo debe ser utilizado individualmente.

Pero sigue siendo una buena alternativa, ya que el algoritmo ya viene implementado en la propia herramienta de Unity, por lo que es sencilla de manejar y se da por hecho que debe funcionar correctamente.

6.- Realidad Virtual en Unity

Nos ha parecido interesante también incorporar la realidad virtual a nuestro proyecto, de manera que, situando una cámara enfrente del coche, el usuario pueda experimentar la realidad virtual como si él mismo estuviera dentro del coche.
Para esto, Unity posee soporte para ser usado junto con dispositivos de realidad virtual.

Unity permite incorporar dispositivos de realidad virtual directamente, facilitando su uso. Además, proporciona una API y un conjunto de características con compatibilidad para varios dispositivos de realidad virtual.

6.1.- Habilitando el soporte de realidad virtual de Unity

Como ya se ha comentado, Unity permite el uso de dispositivos de realidad virtual de manera muy sencilla. Para habilitar el soporte de realidad virtual abriremos las opciones del proyecto (Project Settings).
Dentro de estas opciones nos encontramos con la opción de *XR Settings*. XR es un término que abarca realidad vitual (VR), realidad aumentada (AR) y realidad mixta (MR). En esta opción debemos marcar la casilla *Virtual Reality Supported*. Activado el soporte para realidad virtual, usaremos la lista que se nos despliega para agregar o eliminar dispositivos de realidad virtual.
Es importante tener en cuenta que el orden de esta lista será el orden en el que Unity intentará habilitar los dispositivos de realidad virtual en tiempo de ejecución. El primer dispositivo que se inicialice correctamente, será el dispositivo habilitado.

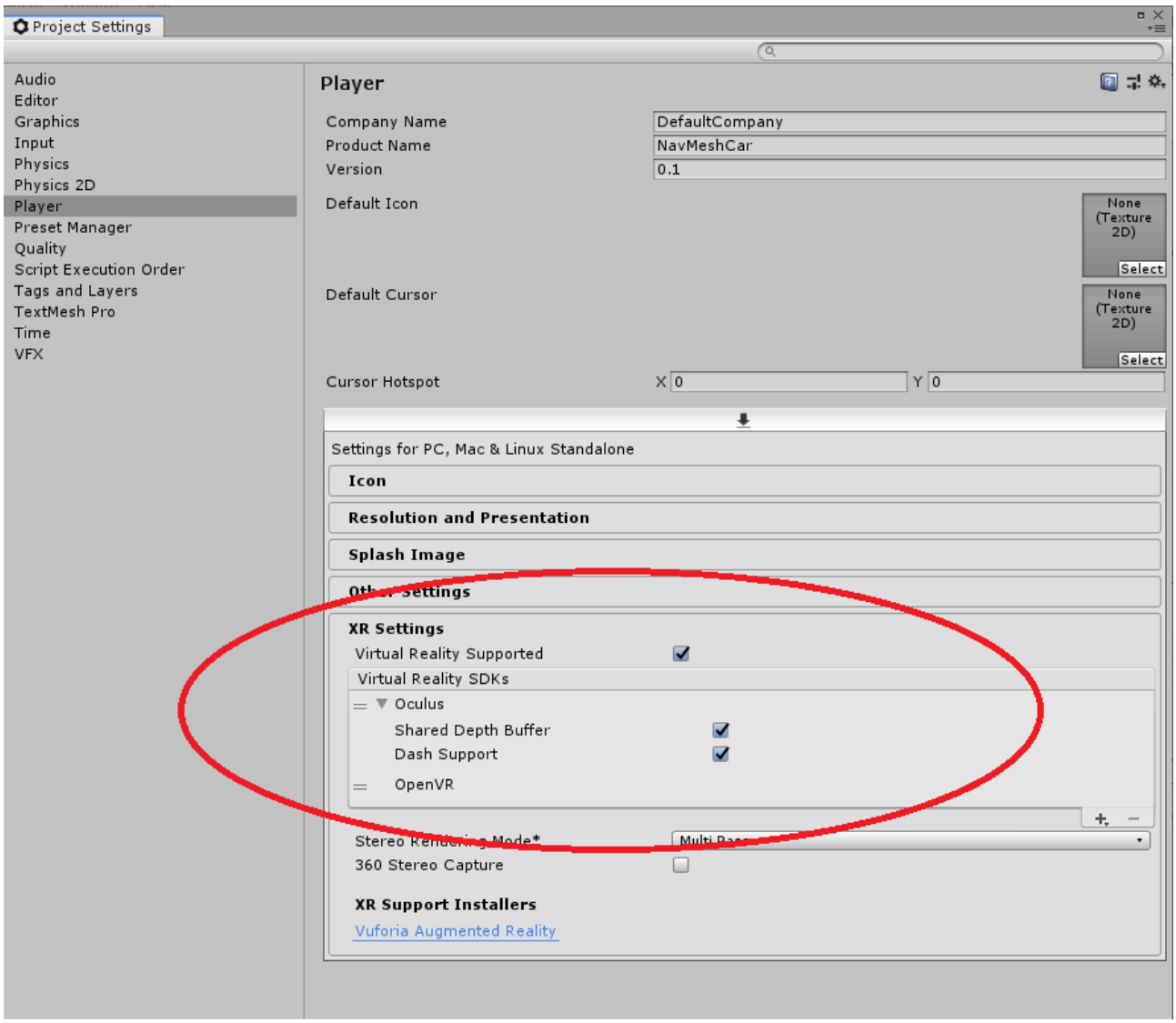

Figura 6.1. Opciones de habilitación de soporte para realidad virtual

Una vez que la realidad virtual está habilitada en Unity, ocurren varias cosas automáticamente:

1.- *Renderización automática a un "head-mounted display"*. Todas las cámaras de la escena automáticamente se renderizan en la pantalla de, en este caso, las gafas de realidad virtual. Las matrices de vista y proyección se ajustan para poder tener en cuenta el movimiento de la cabeza, la posición y el campo de visión.
Para que una cámara en concreto no sea renderizada en las gafas de realidad virtual, es necesario deshabilitar la opción *Target Eye*. También permite la visualización en pantalla de un único ojo.

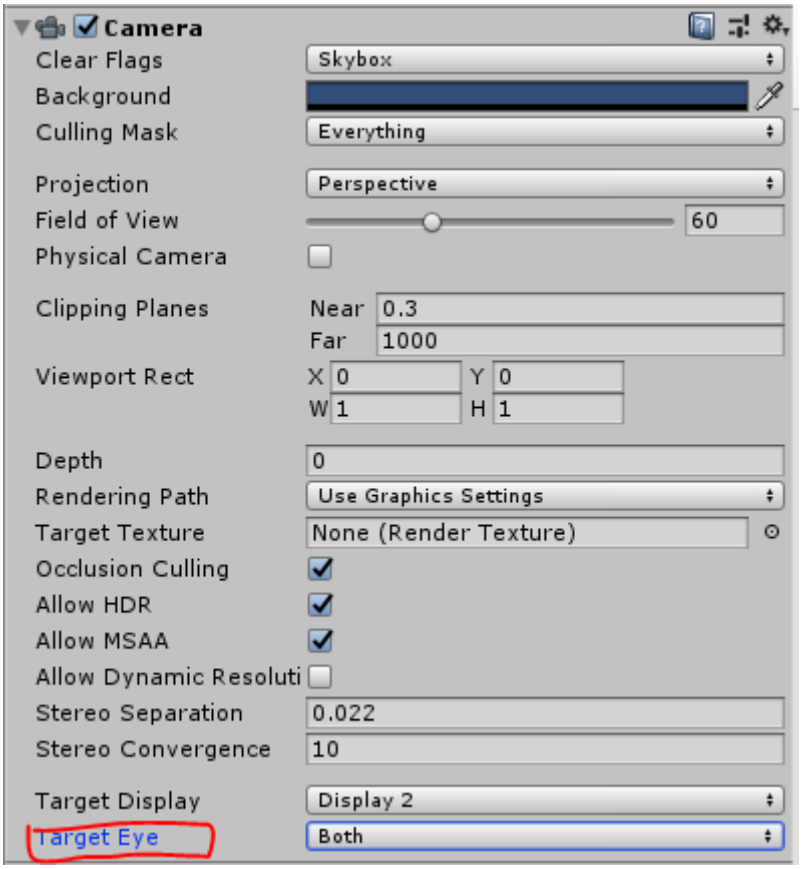

Figura 6.2. Opciones de la cámara

2.- *Input automático de seguimiento de la cabeza*. El seguimiento de la cabeza y la posición en el espacio se aplican automáticamente para que la posición y la orientación coincida con la posición y la orientación del usuario. Esto proporciona una buena experiencia de realidad virtual.
3.- *Propiedad transform de la cámara.* La propiedad Transform de la cámara se anula automáticamente, ya que la posición de dicha cámara vendrá dada por la posición de la cabeza del usuario.

Además, para que la experiencia sea plena, Unity ofrece una serie de recomendaciones tanto de software como de hardware.

6.2.- Recomendaciones de software y hardware para realidad virtual

En cuanto al hardware, para lograr una buena experiencia de realidad virtual, es muy importante lograr una tasa de frames similar a la del dispositivo que utiliza el usuario. Para ello, esta tasa debe coincidir con la tasa de refresco o actualización de la pantalla del dispositivo de realidad virtual.
La página oficial de Unity proporciona una tabla enumerando las tasas de refresco para dispositivos de realidad virtual comunes:

VR Device	Refresh Rate
Gear VR	60hz
Oculus Rift	90hz
Vive	90hz

Figura 6.3. Tabla de tasas de refresco de diferentes dispositivos VR

En cuanto al software, nuevamente la página oficial de Unity ofrece una lista de recomendaciones.

- **Windows**: Windows 7, 8, 8.1, y Windows 10 son todos compatibles.
- **Android**: Android OS Lollipop 5.1 o superior.
- **OS X**: OSX 10.9+ con Oculus 0.5.0.1 runtime. Sin embargo, Oculus ha pausado el desarrollo para OS X, entonces utilice Windows para una funcionalidad VR nativa en Unity.
- **Graphics card drivers**: Asegúrese de que sus drivers (controladores) estén actualizados. Cada dispositivo se mantiene al día con los drives más nuevos, por lo que los drivers más viejos podrían no estar soportados.

Figura 6.4. Recomendaciones de Unity respecto al software para realidad virtual

7.- CONCLUSIONES

La finalidad de este proyecto era contestar a la pregunta “¿es viable usar Unity para definir algoritmos de machine learning para selección de ruta óptima en conducción autónoma?”
Y bien, tal y como hemos visto, sí es posible definir y utilizar algoritmos de machine learning de selección de ruta óptima en Unity. De hecho, en Unity se pueden definir este tipo de algoritmos y cualquier tipo de algoritmo que se pueda ocurrir.
Para eso, ya hemos visto que existe la librería llamada ml-agents, que utiliza una red neuronal para entrenar a los agentes. El usuario sólo se debe preocupar de saber qué objetos del entorno debe tener en cuenta el agente (los inputs de la red neuronal) y qué acciones deben dar un tipo de recompensa u otro. La red neuronal, en nuestro caso el algoritmo de tensorflow, se encargará de, con estos datos, ir aprendiendo.

Es posible también pensar en definir una red neuronal desde cero, sin embargo, es mucho más sencillo, rápido y probablemente más efectivo utilizar una red neuronal ya creada y probada.

Otra opción que hemos visto es la inteligencia artificial de navegación que viene con la propia herramienta de Unity. Hemos visto que sí es viable usar este algoritmo, pero es necesario hacer algunas modificaciones para que funcione correcto del todo. Además, este algoritmo sería, en principio, imposible aplicarlo a cualquier otro algoritmo de conducción, ya que este algoritmo de navegación ya se encarga de mover al agente.
En cambio, con la librería de ml-agents, sí se podría aplicar a un algoritmo ya definido de conducción.
Por otra parte, el algoritmo de navegación está muy limitado, mientras que en la librería ml-agents, el programador tiene vía libre a la hora de añadir cosas como que el coche sepa interpretar un stop, por ejemplo.

Dicho todo esto, si se quiere un buen algoritmo de selección de ruta óptima, la mejor opción sin duda es la de la librería ml-agents. Mientras que si se busca un algoritmo más sencillo, es mejor utilizar el algoritmo de navegación.

8.- TRABAJOS FUTUROS

Como consecuencias de este proyecto y sus conclusiones, un posible e interesante trabajo a realizar en un futuro es crear un algoritmo complejo y completo de selección de ruta óptima utilizando la ya vista librería de machine learning ml-agents.
Este algoritmo debería ser capaz de detectar posibles atascos, calles cortadas, señales de tráfico como stop o semáforos, etc.
O que dicho algoritmo se pueda aplicar a un algoritmo ya definido de conducción.

9.- BIBLIOGRAFIA

https://docs.unity3d.com/Manual/nav-InnerWorkings.html
Página oficial de Unity https://unity.com/
ML-Agents https://unity3d.com/es/how-to/unity-machine-learning-agents https://unity3d.com/es/machine-learning
Virtual Reality https://docs.unity3d.com/es/current/Manual/VROverview.html
Redes Neuronales https://www.xataka.com/robotica-e-ia/las-redes-neuronales-que-son-y-por-que-estan-volviendo
https://es.wikipedia.org/wiki/Red_neuronal_artificial