Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Máster

MBITI WORLD PARA UNITY MACHINE LEARNING AGENTS

Alumno/a: **Sánchez Vílchez, José David**

Tutor/a: Prof. D. Víctor Manuel Rivas Santos
Dpto.: Informática

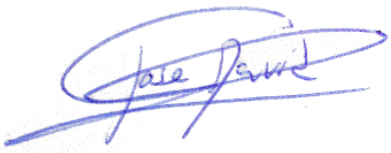
Junio, 2018

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Víctor Manuel Rivas Santos, tutor del Proyecto Fin de Carrera titulado: mBiti World para Unity Machine Learning Agents, que presenta José David Sánchez Vílchez, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2018

El alumno:



José David Sánchez Vílchez

El tutor:

RIVAS SANTOS VICTOR MANUEL - 29798183G

Firmado digitalmente por RIVAS SANTOS VICTOR MANUEL - 29798183G
Fecha: 2018.06.04 10:26:18 +02'00'

Víctor Manuel Rivas Santos

AGRADECIMIENTOS

Me gustaría expresar mi agradecimiento a todas las personas que han hecho posible el desarrollo de este trabajo:

A mis padres, por su apoyo durante este tiempo.

A los profesores, por el conocimiento transmitido en sus asignaturas.

En especial a mi tutor, por guiarme a lo largo del proyecto, resolver mis dudas y supervisar mi trabajo.

ÍNDICE

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Introducción al proyecto

El objetivo de este TFM es recrear usando Unity y ML-Agents un antiguo proyecto denominado "mBiti World", en el cual distintos agentes inteligentes convivían en un mundo artificial intentando aprender a sobrevivir.

Los mBitis pueden ser herbívoros o carnívoros. Pueden moverse, comer y morir, debiendo aprender la mejor estrategia para sobrevivir. Además, pueden reproducirse de forma que evolucionen y transmitan a las siguientes generaciones nuevas características.

El producto final será una aplicación (juego) que permita indicar unos parámetros iniciales de las distintas poblaciones que puedan existir, tratando de llegar a un estado de equilibrio entre ellas.

El desarrollo de este proyecto será llevado a cabo mediante el motor de juegos Unity y el plugin ML-Agents, que permite crear agentes inteligentes mediante técnicas de *machine learning*.

1.2. Motivación

En los últimos años, el aprendizaje automático (*machine learning*) está cambiando la manera en que se obtiene comportamiento inteligente de agentes autónomos. Mientras que en el pasado el comportamiento solía estar determinado por un algoritmo preestablecido, cada vez es más frecuente que se le enseñe al agente (ya sea un robot o un avatar virtual) a través de la interacción en un entorno de entrenamiento. Este método se utiliza para crear comportamiento inteligente para robots industriales, drones, vehículos autónomos o hasta personajes de videojuegos [1].

Ilustración 1. Interpretación de datos mediante *machine learning* en conducción autónoma

La calidad de este entorno de entrenamiento es fundamental para obtener el comportamiento deseado. El escenario típico para entrenar agentes en entornos virtuales consiste en tener un entorno y un agente que estén estrechamente relacionados. Las acciones del agente cambian el estado del entorno y proporcionan recompensas al agente. En esto se basa el ciclo de entrenamiento del aprendizaje reforzado.

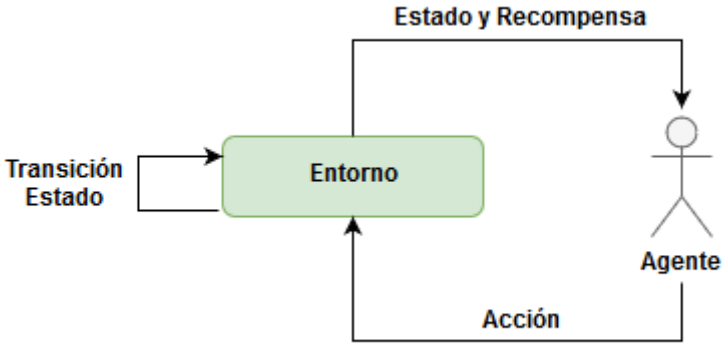


Ilustración 2. Ciclo de entrenamiento en aprendizaje reforzado

A finales de 2017 Unity publicó un sistema que permite utilizar *machine learning* para desarrollar agentes inteligentes. Este sistema se denominó Machine Learning Agents (ML-Agents).

ML-Agents es un plugin para Unity de código abierto que permite que juegos y simulaciones funcionen como entornos para el entrenamiento de agentes inteligentes. Los agentes se entrenan utilizando aprendizaje reforzado mediante una API de

Python. ML-Agents proporciona implementaciones (basadas en TensorFlow) de algoritmos que permiten entrenar agentes inteligentes para juegos [2].

Estos agentes entrenados se pueden utilizar para fines múltiples, incluido el control de comportamiento de NPC (personajes no jugadores) en una variedad de configuraciones, como multiagente y adversarial, pruebas automatizadas de juegos y evaluación de diferentes decisiones de diseño.

ML-Agents está indicado tanto para desarrolladores de videojuegos como para investigadores de inteligencia artificial, ya que proporciona una plataforma en la que se pueden evaluar distintos avances en IA y tras esto ponerse a disposición de comunidades más amplias de investigación y desarrollo de videojuegos.

2. OBJETIVOS

Los objetivos del TFM son los siguientes:

- Describir las posibilidades que brinda la plataforma ML-Agents a la hora de usar *machine learning* en aplicaciones UNITY
- Definir las necesidades de la aplicación a crear
- Realizar un estudio de los algoritmos implementados en ML-Agents susceptibles de ser usados para este TFM
- Emplear una metodología basada en técnicas de Ingeniería del Software para el establecimiento y documentación de los requisitos de las aplicaciones, metodología de diseño e implementación, así como prueba.
- Crear un prototipo de la aplicación donde se muestren las funcionalidades consideradas más prioritarias al documentar los requisitos.
- Redactar la memoria que recoja todo el trabajo desarrollado, así como los manuales de instalación y usuario.

3. ESTADO DEL ARTE

En este apartado se explica en que consiste el aprendizaje automático y las diferentes técnicas existentes. Tras esto se describe el método que ML-Agents utiliza.

3.1. Machine learning

El aprendizaje automático (*machine learning*) es una rama de la inteligencia artificial que se centra en el aprendizaje de patrones a partir de datos. Proporciona a un sistema la capacidad de aprender a partir de datos de forma autónoma (en lugar de hacerlo de forma prediseñada) utilizando información y observaciones para encontrar patrones y hacer predicciones sobre resultados futuros. Es decir, el sistema debe aprender la relación apropiada entre entradas y salidas. De esta forma, el sistema podrá elegir la mejor acción a realizar en cada momento para optimizar su resultado.

Existen diferentes formas de realizar esto, dependiendo del tipo de datos a partir del cual se realiza el aprendizaje.

Las tres clases principales de algoritmos de aprendizaje automático son: aprendizaje no supervisado, aprendizaje supervisado y aprendizaje reforzado [3].

3.1.1. Aprendizaje no supervisado

El objetivo del aprendizaje no supervisado es agrupar elementos similares en un conjunto de datos. Esta agrupación se realiza según sus atributos. Primero, se definirían los atributos básicos de los elementos para luego proporcionar este conjunto de datos a un algoritmo de aprendizaje no supervisado y especificar la cantidad de grupos. El algoritmo dividiría el conjunto de datos en diferentes grupos de tal forma que los elementos de cada grupo serían similares entre sí.

Con el aprendizaje no supervisado no se proporcionan ejemplos específicos. Simplemente se definen los atributos apropiados y se depende del algoritmo para descubrir los grupos por sí mismo.

3.1.2. Aprendizaje supervisado

El objetivo del aprendizaje supervisado no es solo agrupar elementos similares, sino directamente aprender un mapeo de cada elemento al grupo (o clase) al que pertenece. Estos algoritmos aceptan etiquetas de entrada además de atributos. Se proporciona un conjunto de datos (atributos y etiqueta para cada elemento) a un algoritmo de aprendizaje supervisado que aprendería un mapeo de los atributos del elemento a una etiqueta. Al modelo resultante se le podría proporcionar los atributos de un nuevo elemento y estimaría una etiqueta para ese elemento.

Tanto para el aprendizaje supervisado como para el no supervisado, hay dos tareas que deben realizarse: la selección del atributo y la selección del modelo. La selección de atributos (también llamada selección de características) se refiere a seleccionar cómo se quiere representar a la entidad de interés. La selección del modelo, por otro lado, se refiere a seleccionar el algoritmo (y sus parámetros) que realice bien la tarea. Ambas son áreas de investigación en aprendizaje automático y, en la práctica, requieren realizar pruebas para lograr un buen rendimiento.

3.1.3. Aprendizaje reforzado

El aprendizaje reforzado se puede considerar como una forma de aprendizaje para la toma de decisiones de forma secuencial que comúnmente se asocia con el control de robots. En un momento dado, el robot (agente inteligente) percibe el entorno a través de sus sensores, procesa esta información y produce una acción. Es decir, toma continuamente decisiones sobre cómo interactuar en este entorno dada su visión del mundo y su objetivo.

Concretamente, el objetivo del aprendizaje reforzado es aprender una política, que es básicamente un mapeo de observaciones a acciones. Una observación sería lo que el agente puede percibir de su entorno y una acción sería un cambio en la configuración del agente.

Otro elemento de la tarea de aprendizaje reforzado es la señal de recompensa. Cuando se entrena a un agente se le proporcionan recompensas (positivas y negativas) que indican cómo de bien lo está haciendo para completar la tarea. El hecho de que las recompensas sean escasas (es decir, que no se proporcionen en

cada paso, sino solo cuando un agente llega a una situación de éxito o fracaso) es una característica distintiva del aprendizaje reforzado y es la razón por la que aprender buenas políticas puede ser difícil y consumir mucho tiempo para entornos complejos.

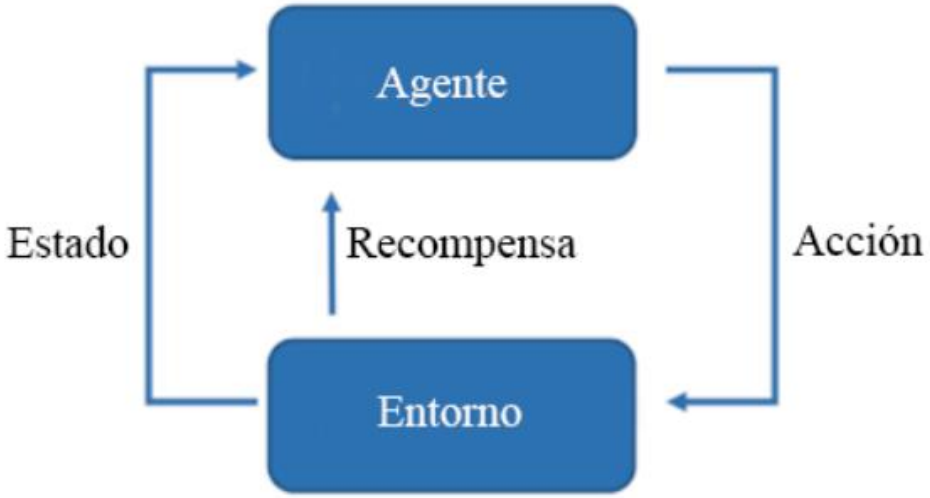


Ilustración 3. Ciclo del aprendizaje reforzado

Aprender una política generalmente requiere muchas pruebas y actualizaciones iterativas de políticas. Específicamente, el agente se pone en diferentes situaciones y, con el tiempo, aprende una política óptima que le permite conseguir su objetivo de la manera más efectiva.

De modo similar al aprendizaje supervisado y no supervisado, el aprendizaje reforzado también implica dos tareas: selección de atributos y selección de modelos. La selección de atributos consiste en definir el conjunto de observaciones, acciones y recompensas que mejor ayuda al agente a completar su objetivo. Mientras que la selección del modelo consiste en la forma de definir la política (asignación de observaciones a acciones) y sus parámetros. En la práctica, el entrenamiento de comportamientos es un proceso iterativo que puede requerir el cambio de atributos y modelos.

Un aspecto común de las tres ramas del aprendizaje automático es que todas involucran una fase de entrenamiento y una fase de inferencia, además de selecciones de atributos y modelos. La fase de entrenamiento implica construir un

modelo utilizando los datos proporcionados, mientras que la fase de inferencia implica aplicar este modelo a datos nuevos nunca vistos.

Lo que diferencia a las tres ramas es el tipo de datos disponibles para aprender. En el aprendizaje no supervisado, el conjunto de datos es una colección de atributos; en el aprendizaje supervisado, una colección de pares de atributos y etiquetas; y en el aprendizaje reforzado, una colección de tuplas de observación, acción y recompensa.

3.2. Machine learning en ML-Agents

ML-Agents utiliza aprendizaje reforzado. Como se ha mencionado previamente, el aprendizaje reforzado es un método de aprendizaje automático basado en no decirle al sistema qué hacer, sino solo lo que está bien y lo que está mal. De esta forma, el sistema comprenderá que tiene que realizar ciertas acciones para obtener una recompensa positiva.

En ML-Agents, el modelo utilizado en el aprendizaje reforzado es una red neuronal, un sistema informático basado en el sistema nervioso.

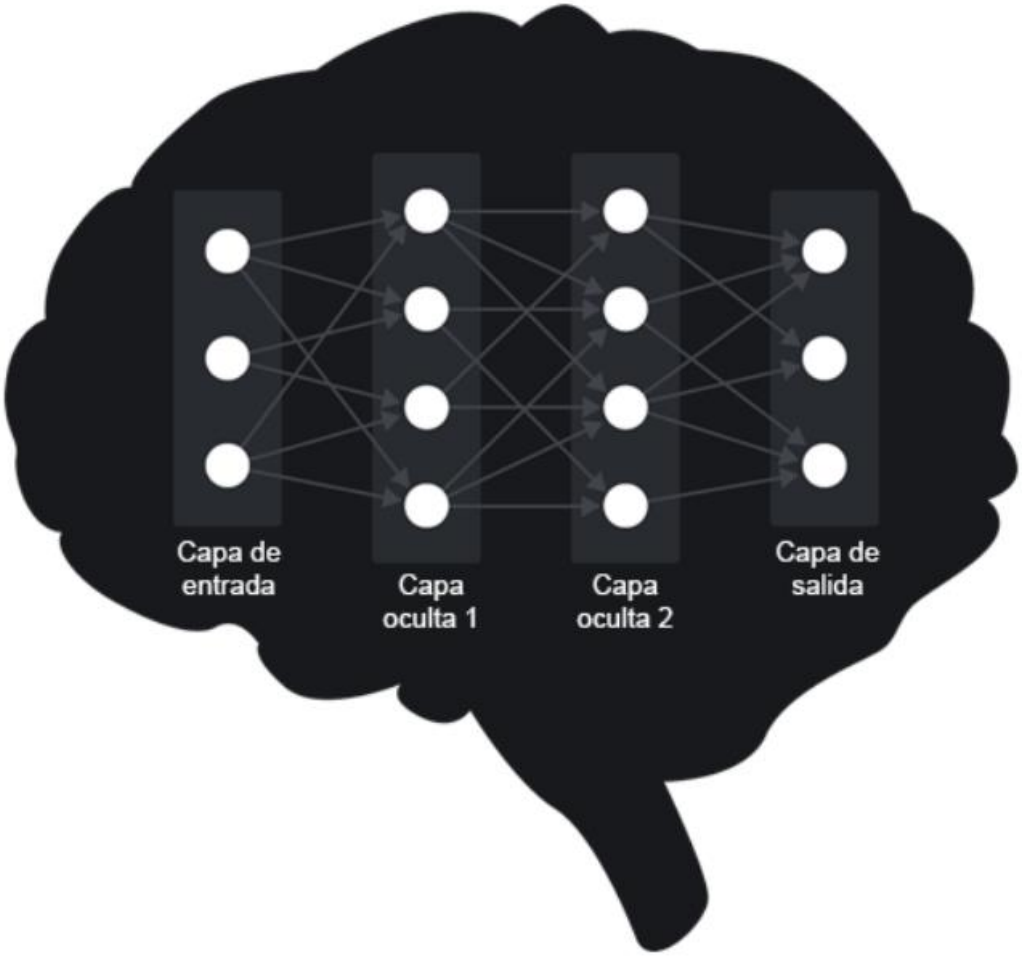


Ilustración 4. Red neuronal

Las redes neuronales están estructuradas en unidades o "neuronas", que se agrupan en capas. La capa que interactúa con el entorno externo y recopila toda la información de entrada es la capa de entrada. En el lado opuesto, la capa de salida contiene las neuronas que almacenan información sobre el resultado de una determinada entrada en la red. Entre estas se sitúan las capas ocultas, que contienen las neuronas que realizan los cálculos. Aprenden representaciones abstractas y complejas de los datos de entrada, lo que permite que sus resultados sean finalmente "inteligentes". La mayoría de las capas están conectadas por completo, es decir, las neuronas de cada capa están conectadas a todas las neuronas de la capa anterior. Cada conexión se define por un "peso", que es un valor numérico que puede fortalecer o debilitar el vínculo entre las neuronas [4].

Los agentes utilizan este modelo de aprendizaje reforzado. Realizan acciones en un entorno y estas causan un cambio en el entorno, que se proporciona al agente,

junto con alguna recompensa o castigo. La acción ocurre en el entorno de aprendizaje, que en términos prácticos corresponde a una escena de Unity.

Dentro del entorno de aprendizaje hay una “academia”, que es una secuencia de comandos que define las propiedades del entrenamiento. La academia pasa estos parámetros a los “cerebros”, que son entidades que contienen el modelo a entrenar. Finalmente, los agentes están conectados a los cerebros, de los que obtienen las acciones y a los que proporcionan información para facilitar el proceso de aprendizaje.

Para realizar el entrenamiento, el sistema usa un módulo adicional que permite que los cerebros se comuniquen con un entorno externo de Python utilizando la biblioteca de *machine learning* de TensorFlow. Una vez que se completa el entrenamiento, este entorno sintetiza el proceso de aprendizaje en un modelo (red neuronal), un archivo binario que se reimportará a Unity para ser utilizado como un cerebro entrenado.

En la sección de implementación se explicará más a fondo el funcionamiento de ML-Agents.

4. TECNOLOGÍAS UTILIZADAS

4.1. Unity

Unity es un motor de videojuegos (*game engine*) utilizado principalmente para el desarrollo de videojuegos y simulaciones en 3D para ordenador, videoconsolas y dispositivos móviles [5]. Una de sus características principales es la programación de scripts en C#.

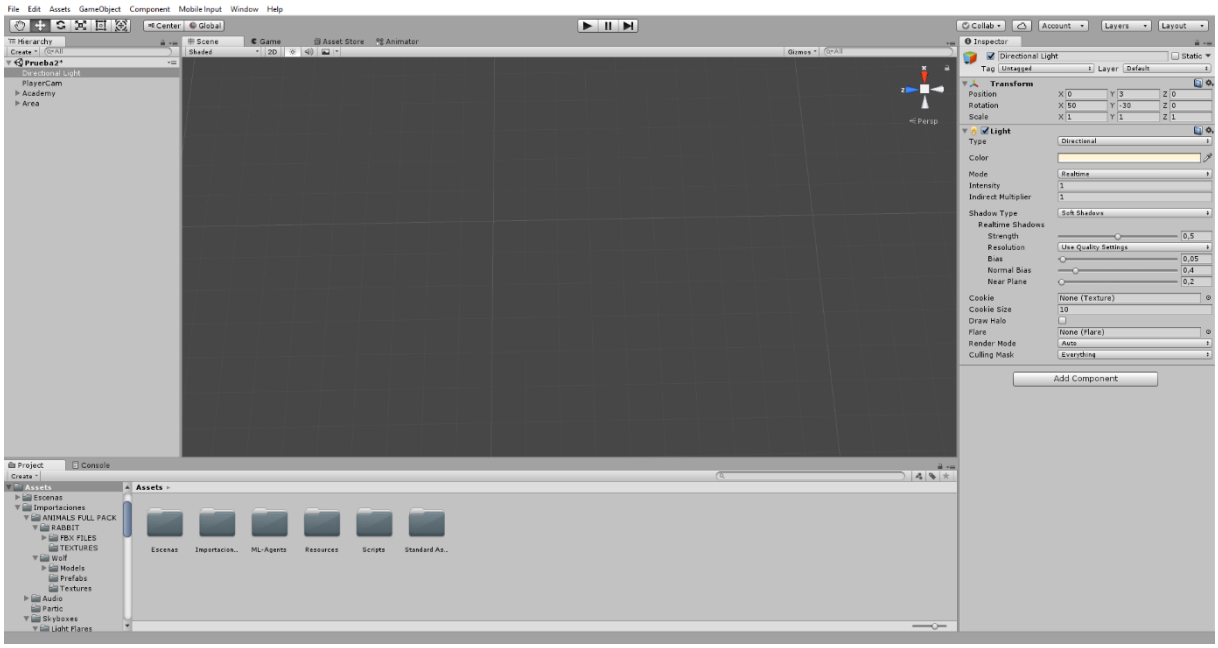


Ilustración 5. Interfaz de Unity

Unity dispone de una versión gratuita para estudiantes y aficionados. Se puede descargar desde el siguiente enlace:

https://unity3d.com/es/get-unity/download

4.2. MonoDevelop

MonoDevelop es el entorno de desarrollo integrado (IDE) proporcionado con Unity.

Se instala al mismo tiempo que Unity.

4.3. ML-Agents

ML-Agents es un plugin para Unity de código abierto que permite que juegos y simulaciones funcionen como entornos para el entrenamiento de agentes inteligentes.

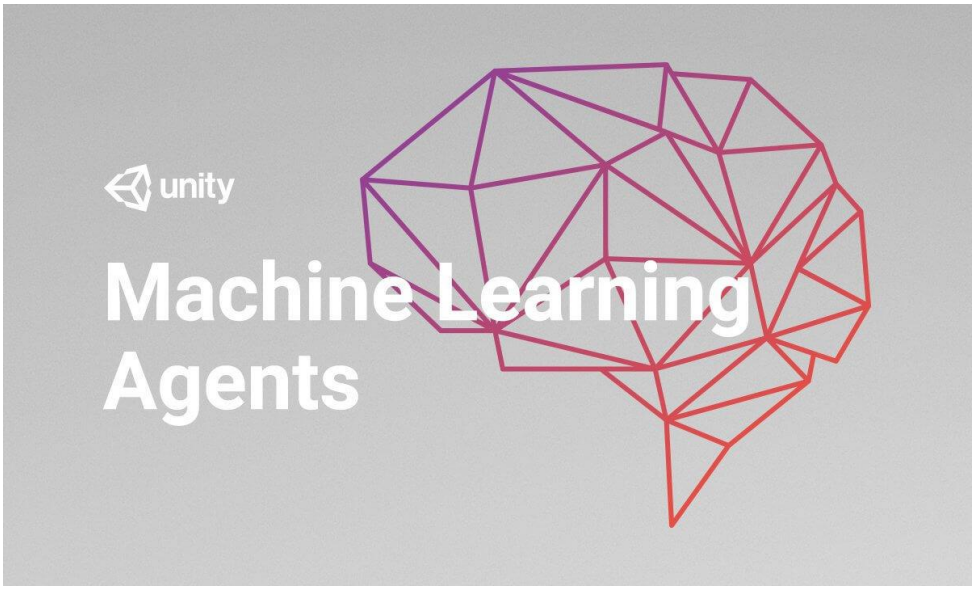


Ilustración 6. Logo ML-Agents

En el momento del desarrollo de este proyecto ML-Agents se encuentra en versión beta y sigue en construcción. Se puede descargar en la siguiente página:

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

4.4. Jupyter Notebook

Jupyter Notebook es una aplicación web de código abierto que permite editar y ejecutar documentos que contengan código, ecuaciones, visualizaciones y textos narrativos. Los usos incluyen: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización de datos y *machine learning*.

Jupyter Notebook es necesario para ejecutar el entrenamiento externo en ML-Agents.

Para poder instalarlo hay que descargar Anaconda.

4.5. Anaconda

Anaconda es una versión libre de *Python* que instala los paquetes más importantes para tratamiento de grandes volúmenes de datos, cálculos científicos y análisis de predicciones.

Anaconda se distribuye con licencia BSD. Se puede descargar desde siguiente enlace:

https://www.anaconda.com/download/

4.6. Microsoft Word

Microsoft Word es el procesador de textos que he utilizado para elaborar la memoria.

Microsoft Word ya lo tenía instalado en el ordenador.

4.7. Visual Paradigm

Visual Paradigm es una herramienta CASE (Ingeniería de Software Asistida por Computadora). Permite elaborar diferentes tipos de diagramas UML: de clases, de casos de uso, de secuencia, de comunicación, de máquina de estados, de actividad, de componentes, de despliegue, de paquetes, de objeto, de estructura, de tiempos y de interacción [6].

Visual Paradigm la he utilizado para realizar los diagramas UML de los apartados de análisis y diseño.

He utilizado la versión con licencia para estudiantes de la universidad de Jaén. También dispone de una versión de prueba de 30 días:

https://www.visual-paradigm.com/download/

4.8. XMind

XMind es un programa para realizar mapas conceptuales. Se suele utilizar en la gestión de proyectos para recoger ideas, organizar la información y colaborar en equipo. Aparte de mapas mentales, permite realizar diagramas causa-efecto, diagramas de árbol, organigramas, mapas de red y hojas de cálculo entre otros [7].

Lo he utilizado para crear el RBS.

XMind dispone de una versión gratuita y otra de pago con funcionalidad adicional:

https://www.xmind.net/download/win/

4.9. Draw.io

Draw.io es un software en línea gratuito para hacer diagramas de flujo, diagramas de proceso, organigramas, UML, E-R y diagramas de red.

Lo he utilizado para crear el modelo E-R y las imágenes de las metodologías.

Se puede acceder a Draw.io mediante el siguiente enlace:

https://www.draw.io/

4.10. Microsoft Project

Microsoft Project es un software de administración de proyectos que he utilizado para crear el diagrama de Gantt.

Es una aplicación de pago, pero tiene una versión de prueba:

https://products.office.com/es/project/project-professional-desktop-software

4.11. Photoshop

Photoshop es el editor de gráficos que he utilizado para modificar imágenes.

Photoshop tiene una versión de prueba de 30 días:

https://www.adobe.com/es/products/photoshop/free-trial-download.html

4.12. Nvidia ShadowPlay

Nvidia ShadowPlay es una herramienta de grabación de pantalla que cuenta con aceleración por hardware usando una GPU GeForce.

La he utilizado para grabar el vídeo de la aplicación.

Nvidia ShadowPlay es gratuita y se puede descargar desde el siguiente enlace:

https://www.nvidia.com/en-us/geforce/geforce-experience/shadowplay/

4.13. Adobe Premiere

Adobe Premiere es el software de edición que he utilizado para editar el vídeo de la aplicación.

Adobe Premiere tiene una versión de prueba de 30 días.

https://www.adobe.com/es/products/premiere/free-trial-download.html

5. METODOLOGÍA

5.1. Selección de la metodología

Para elegir la metodología a utilizar en este proyecto he tenido que tener en cuenta el alcance, los plazos, los riesgos y los objetivos.

Los requisitos no son estables, podrían cambiar a lo largo del proyecto, o añadirse nuevos. Los plazos son flexibles al no tener una fecha límite determinada. Además, al utilizar en este proyecto una herramienta por primera vez, podrían surgir problemas que alteren lo planificado.

Por estas razones, no conviene el modelo en cascada, ya que este no se adapta a los cambios de requisitos, no se obtiene ninguna versión del producto hasta el final y no acomoda posibles riesgos.

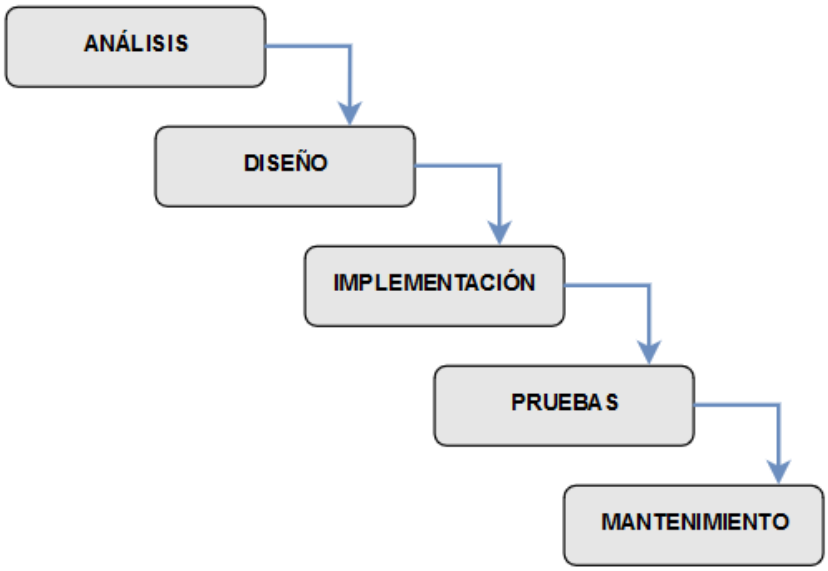


Ilustración 7. Fases del modelo en cascada

En cambio, el modelo incremental podría ser más conveniente para este proyecto.

En el modelo incremental el producto se divide en incrementos que producen entregas de producto operativo. Los incrementos se estructuran de modo que los primeros incluyan los requisitos más importantes y cada uno de los incrementos

contiene las fases de análisis, diseño, implementación y pruebas. Una vez comenzado un incremento, los requisitos no se modifican hasta que este ha terminado, aunque los cambios a una parte del producto se pueden incluir en un incremento posterior.

Sin embargo, este modelo requiere una definición clara y completa del sistema para poder dividirse en incrementos. Los requisitos del sistema deben estar claramente definidos al principio [8].

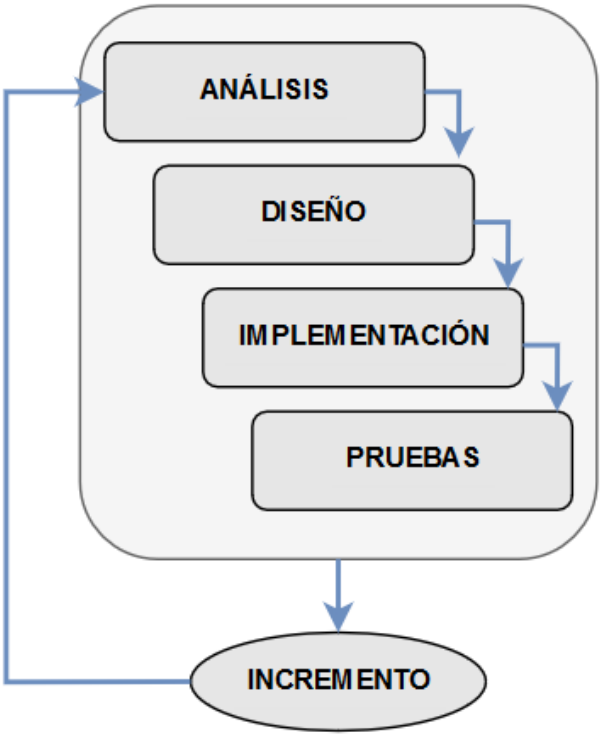


Ilustración 8. Modelo incremental

Finalmente he decidido utilizar una metodología ágil.

En un proceso ágil se desarrolla el software de forma iterativa con mucho énfasis en las actividades de construcción. Algunos requisitos persisten y otros cambian, al igual que sus prioridades [9].

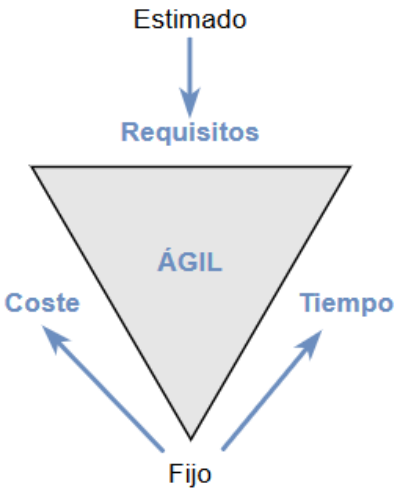


Ilustración 9. Triángulo ágil con las variables de un proyecto

Además, me permite utilizar los incrementos de software como prototipos para que el tutor proporcione feedback [10].

De entre las diferentes metodologías ágiles, he decidido utilizar Scrum.

5.2. Definición de la metodología

Scrum es un proceso ágil para administrar el trabajo con énfasis en el desarrollo de software.

Un principio clave de Scrum es la comprensión de que los clientes cambiarán de opinión sobre lo que quieren o necesitan y que habrá desafíos impredecibles, para los cuales un enfoque predictivo o planificado no es adecuado. Por ello, Scrum acepta que el problema no puede ser completamente comprendido o definido de antemano, y en su lugar se centra en cómo maximizar la capacidad de un equipo de trabajo para realizar entregas frecuentes y responder a requisitos emergentes [11].

En Scrum el trabajo se divide en acciones que se pueden completar en iteraciones de tiempo fijo, llamados **sprints**. Su duración se fija por adelantado para cada sprint y normalmente es de dos semanas [12].

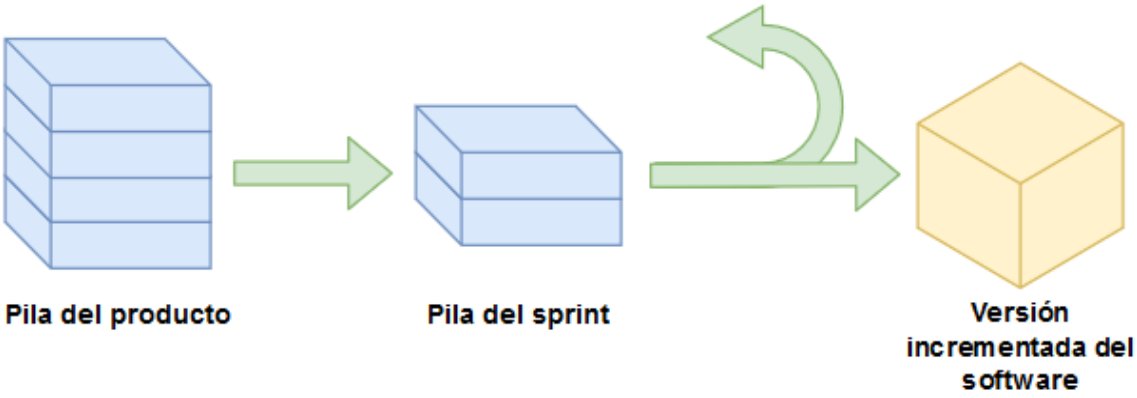


Ilustración 10. Ciclo de trabajo en Scrum

El ciclo de trabajo en Scrum consiste en lo siguiente:

1. Se definen los requisitos y para cada requisito principal se crea un bloque de trabajo, llamado **historia** de usuario.
2. Se ordenan las historias en una **pila** del producto (*backlog*) según su prioridad de entrega.
3. Se toma un grupo de historias, con el que se trabaja durante un **sprint** (pila del sprint).
4. Una vez finalizado un sprint se entrega al cliente el resultado del trabajo. Tras esto se vuelve al punto 2 hasta terminar la pila del producto.

6. ANÁLISIS

6.1. Análisis de requisitos

A continuación se definen los requisitos para la aplicación:

6.1.1. Requisitos funcionales

- **Desplazamiento**. Los animales se podrán mover por el entorno con una velocidad determinada.
- **Energía**. Los animales dispondrán de un sistema de energía. Al desplazarse consumirán energía, así como al reproducirse, y la repondrán alimentándose. Si se quedan sin energía morirán.
- **Comer**. Los animales podrán alimentarse de comida. Los depredadores se alimentarán de los herbívoros y estos se alimentarán de plantas del entorno.
- **Visión**. Los animales no saben inicialmente la posición de su alimento, por lo que tendrán que moverse y podrán conocer la posición de este una vez que haya entrado en su rango de visión.
- **Reproducción**. Los animales se podrán reproducir, generando descendencia a la que le transmitirán sus características.
- **Muerte**. Un animal morirá si se queda sin energía o es comido por un depredador.
- **Poblaciones**. Existirán poblaciones de animales herbívoros y carnívoros.
- **IA**. La inteligencia artificial de los animales deberá crearse con ML-Agents.
- **Obstáculos**. En el entorno habrá obstáculos con los que podrán colisionar los animales.
- **Parámetros**. El jugador podrá indicar unos parámetros iniciales de las distintas poblaciones para intentar llegar a un estado de equilibrio entre ellas.
- **Pausa**. El jugador podrá detener el juego en cualquier momento.
- **Persistencia**. El jugador podrá guardar el estado actual del juego y cargarlo posteriormente.

- **Cámara**. El jugador podrá mover la cámara libremente por el entorno.
- **Entorno**. El tamaño del entorno será redimensionable y sus características estarán determinadas por los parámetros iniciales.
- **Representación**. Los elementos del juego estarán representados por modelos 3D y se incluirán animaciones y sonidos.
- **Interfaces**. El juego dispondrá de un menú de inicio, pantalla de configuración de ajustes y menú de pausa.
- **Información de juego**. Durante la partida se mostrará información sobre las poblaciones.
- **Resultados**. Al finalizar el juego se mostrará una pantalla de resultados que indique los cambios en las poblaciones y el tiempo transcurrido.

6.1.2. Requisitos no funcionales

- Los tiempos de carga de la aplicación deben ser inferiores a 5 s.
- El tiempo de aprendizaje de un nuevo usuario debe ser inferior a los 5 min.
- La aplicación será desarrollada para PC.
- La aplicación final debe ocupar menos de 2 GB.
- La aplicación deberá consumir menos de 500 MB de memoria RAM.
- La aplicación debe proporcionar mensajes de error que sean informativos.
- La aplicación debe poseer interfaces gráficas que faciliten su comprensión, aprendizaje y uso.
- La interfaz gráfica deberá poseer elementos interactivos fácilmente identificables.

Debido al modelo de proceso seguido, no todos los requisitos se incluyeron al inicio del proyecto, sino que algunos se añadieron en iteraciones posteriores.

Para reorganizar los requisitos y obtener una visión más clara de estos, realicé el RBS (Requirements Breakdown Structure). El RBS permite identificar los requisitos de alto nivel que son imprescindibles para llevar a cabo el proyecto con éxito.

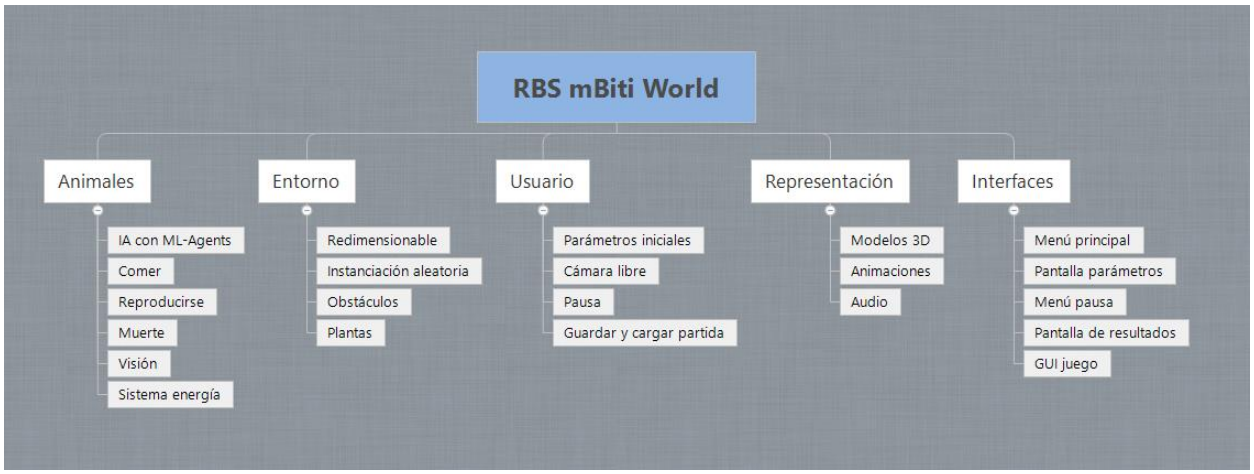


Ilustración 11. RBS

6.2. Historias de usuario

Tras definir los requisitos puedo interpretar las historias de usuario.

A la hora de detallar las historias de usuario se utilizan tarjetas [13], que contienen los siguientes elementos:

- ID: identificador de la historia de usuario.
- Título: título descriptivo de la historia de usuario.
- Descripción: descripción sintetizada de la historia de usuario
- Estimación: estimación del coste de implementación en unidades de desarrollo.
- Prioridad: prioridad en la implementación de la historia de usuario respecto al resto de las historias de usuario.
- Dependencias con otras historias.
- Criterios de aceptación: descripción de las expectativas sobre la aplicación final.

Para la priorización de las historias utilizaré la técnica MoSCoW [14]. Para la estimación utilizaré una escala basada en una sucesión de Fibonacci, pero en este caso con los números 0, 1, 2, 3, 5, 8, 13 y 20, que representan niveles de esfuerzo crecientes.

Ilustración 12. Priorización MoSCoW

La siguiente tabla contiene un resumen de las historias de usuario con su estimación de coste, prioridad y dependencias:

ID	Título	Estimación	Prioridad	Dependencias
1	Desplazamiento	8	Must	
2	Comer	3	Must	
3	Sistema de energía	5	Must	
4	Visión	13	Must	
5	Reproducción	13	Must	
6	Muerte	3	Must	
7	Poblaciones	8	Must	1,2,3,4,5,6
8	Entorno	8	Must	7
9	IA	20	Must	8
10	Parámetros	13	Must	9
11	Pausa	13	Could	
12	Persistencia	20	Could	10
13	Cámara	5	Should	
14	Representación	13	Should	8
15	Interfaces	13	Should	10
16	Información juego	5	Could	10
17	Resultados	8	Could	10

Tabla 1. Historias de usuario

A continuación se muestran las tarjetas de las historias de usuario:

Desplazamiento	
Como usuario quiero que los animales puedan desplazarse para poder obtener comida.	
Criterios de aceptación	**Prioridad**
• Los animales se moverán a una velocidad determinada. • Podrán colisionar con otros objetos.	Must **Estimación** 8

Tabla 2. Historia de usuario 1

Comer	
Como usuario quiero que los animales puedan comer para reponer su energía.	
Criterios de aceptación	**Prioridad**
• Al acercarse lo suficiente a un alimento, el animal lo consumirá. • Al comer repondrá energía.	Must **Estimación** 5

Tabla 3. Historia de usuario 2

Sistema de energía	
Como usuario quiero que los animales tengan un sistema de energía para que necesiten alimentarse para sobrevivir.	
Criterios de aceptación	**Prioridad**
• Los animales consumirán energía al moverse y al reproducirse. • Repondrán energía al alimentarse. • Si se quedan sin energía morirán.	Must **Estimación** 5

Tabla 4. Historia de usuario 3

Visión	
Como usuario quiero que los animales puedan obtener información sobre el entorno para poder llevar a cabo sus objetivos.	
Criterios de aceptación	**Prioridad**
• Los animales tendrán un rango de visión. • Podrán conocer la posición de un objeto una vez que este haya entrado dentro de su rango de visión.	Must **Estimación** 13

Tabla 5. Historia de usuario 4

Muerte	
Como usuario quiero que los animales puedan morir para que necesiten alimentarse para sobrevivir.	
Criterios de aceptación • Los animales morirán si se quedan sin energía, son comidos por un depredador o se acaba su tiempo de vida.	**Prioridad** Must **Estimación** 3

Tabla 6. Historia de usuario 5

Reproducción	
Como usuario quiero que los animales puedan reproducirse para que no se extingan.	
Criterios de aceptación • Los animales transmitirán sus características a su descendencia.	**Prioridad** Must **Estimación** 13

Tabla 7. Historia de usuario 6

Poblaciones	
Como usuario quiero que haya animales herbívoros y carnívoros para estudiar el comportamiento entre ellos.	
Criterios de aceptación • Cada población tendrá sus propias características • Mientras que los herbívoros se alimentan de plantas los carnívoros se alimentan de los primeros. • Las plantas tendrán un periodo de regeneración tras ser comidas.	**Prioridad** Must **Estimación** 8

Tabla 8. Historia de usuario 7

Entorno	
Como usuario quiero que haya un entorno para que los animales puedan convivir en este.	
Criterios de aceptación	**Prioridad**

• En el entorno habrá obstáculos con los que podrán colisionar los animales. • La posición inicial de los elementos del entorno será aleatoria. • El tamaño del entorno será redimensionable.	Must **Estimación** 13

Tabla 9. Historia de usuario 8

IA	
Como usuario quiero que los animales tengan un comportamiento inteligente para poder llevar a cabo sus objetivos.	
Criterios de aceptación	**Prioridad**
• El comportamiento inteligente debe obtenerse mediante ML-Agents. • El objetivo de este comportamiento debe ser que un animal sobreviva el mayor tiempo posible, buscando comida, evitando depredadores si se trata de un herbívoro y evitando colisiones.	Must **Estimación** 20

Tabla 10. Historia de usuario 9

Parámetros	
Como usuario quiero indicar parámetros iniciales de las distintas poblaciones para intentar llegar a un estado de equilibrio entre ellas.	
Criterios de aceptación	**Prioridad**
• Para cada población de animales se indicará el tamaño y características específicas. • El tamaño del entorno dependerá del número de animales establecido.	Must **Estimación** 13

Tabla 11. Historia de usuario 10

Pausa	
Como usuario quiero una funcionalidad de pausa para poder detener el juego en cualquier momento.	
Criterios de aceptación	**Prioridad**
	Could

• Se pausará el juego al pulsar un botón. • Al pausar el juego se mostrará el menú pausa.	**Estimación** 13

Tabla 12. Historia de usuario 11

Persistencia	
Como usuario quiero poder guardar la partida para poder reanudarla posteriormente.	
Criterios de aceptación	**Prioridad**
• Se deben guardar todos los elementos de la partida en curso. • Se guardarán los datos en un fichero. • Se podrá cargar los datos para reanudar la partida.	Could **Estimación** 20

Tabla 13. Historia de usuario 12

Cámara	
Como usuario quiero poder moverme por el entorno para visualizar mejor el juego.	
Criterios de aceptación	**Prioridad**
• Se deben guardar todos los elementos de la partida en curso. • Se guardarán los datos en un fichero. • Se podrá cargar los datos para reanudar la partida.	Should **Estimación** 5

Tabla 14. Historia de usuario 13

Representación	
Como usuario quiero que los animales estén representados por modelos 3D para mejorar la inmersión.	
Criterios de aceptación	**Prioridad**
• Los elementos del entorno estarán representados por modelos 3D. • Habrá terreno con texturas, hierba y un cielo visible. • Los animales tendrán animaciones. • Los animales tendrán sonidos.	Should **Estimación** 13

Tabla 15. Historia de usuario 14

Interfaces	
Como usuario quiero que haya diferentes interfaces para poder interactuar con la aplicación.	
Criterios de aceptación • Habrá un menú de inicio desde el que se podrá iniciar una nueva partida o cargarla. • En el menú de configuración se podrá indicar los parámetros iniciales. • Desde el menú de pausa se podrá guardar y cargar la partida. • Durante el desarrollo de la partida se mostrará información sobre las poblaciones.	**Prioridad** Should **Estimación** 13

Tabla 16. Historia de usuario 15

Información de juego	
Como usuario quiero que se muestre información sobre las poblaciones para visualizar su cambio.	
Criterios de aceptación • Durante el desarrollo de la partida se mostrará el tamaño de las poblaciones.	**Prioridad** Should **Estimación** 5

Tabla 17. Historia de usuario 16

Resultados	
Como usuario quiero que se muestre el resultado de la partida para saber si los parámetros introducidos son adecuados.	
Criterios de aceptación • Se comparará el tamaño inicial y final de las poblaciones. • Se mostrará el tiempo transcurrido.	**Prioridad** Should **Estimación** 8

Tabla 18. Historia de usuario 17

6.3. Planificación

Acorde a la metodología seguida, el trabajo estará dividido en sprints. Cada uno tendrá una duración aproximada de dos semanas.

Al comienzo de cada uno se ordenan las historias en la pila del producto y se toma un grupo de historias con el que se trabaja durante el sprint.

Al principio del proyecto he realizado una planificación inicial según la estimación y prioridad estimadas.

A continuación se pueden ver los sprints con sus historias asociadas:

Sprint	Historia	Estimación	Prioridad
1	Desplazamiento	8	Must
	Comer	3	Must
	Sistema de energía	5	Must
2	Visión	13	Must
	Reproducción	13	Must
3	Muerte	3	Must
	Poblaciones	8	Must
	Entorno	8	Must
4	IA	20	Must
5	IA (Continuación)	20	Must
6	Parámetros	13	Must
	Cámara	5	Should
7	Representación	13	Should
	Interfaces	13	Should
8	Pausa	13	Could
9	Persistencia	20	Could
10	Información juego	5	Could
	Resultados	8	Could

Tabla 19. Sprints

Al final del último sprint se tiene la versión final de la aplicación.

Hay que tener en cuenta que esto representa solo el desarrollo de la aplicación, por lo que no está incluida la investigación previa, el análisis inicial o el desarrollo de la memoria.

Por lo tanto, tras concluir el último sprint, el tiempo de trabajo ha seguido para poder completar la memoria.

6.3.1. Diagrama de Gantt

Una vez realizada la planificación inicial, he creado un diagrama de Gantt para visualizar las dependencias entre las historias.

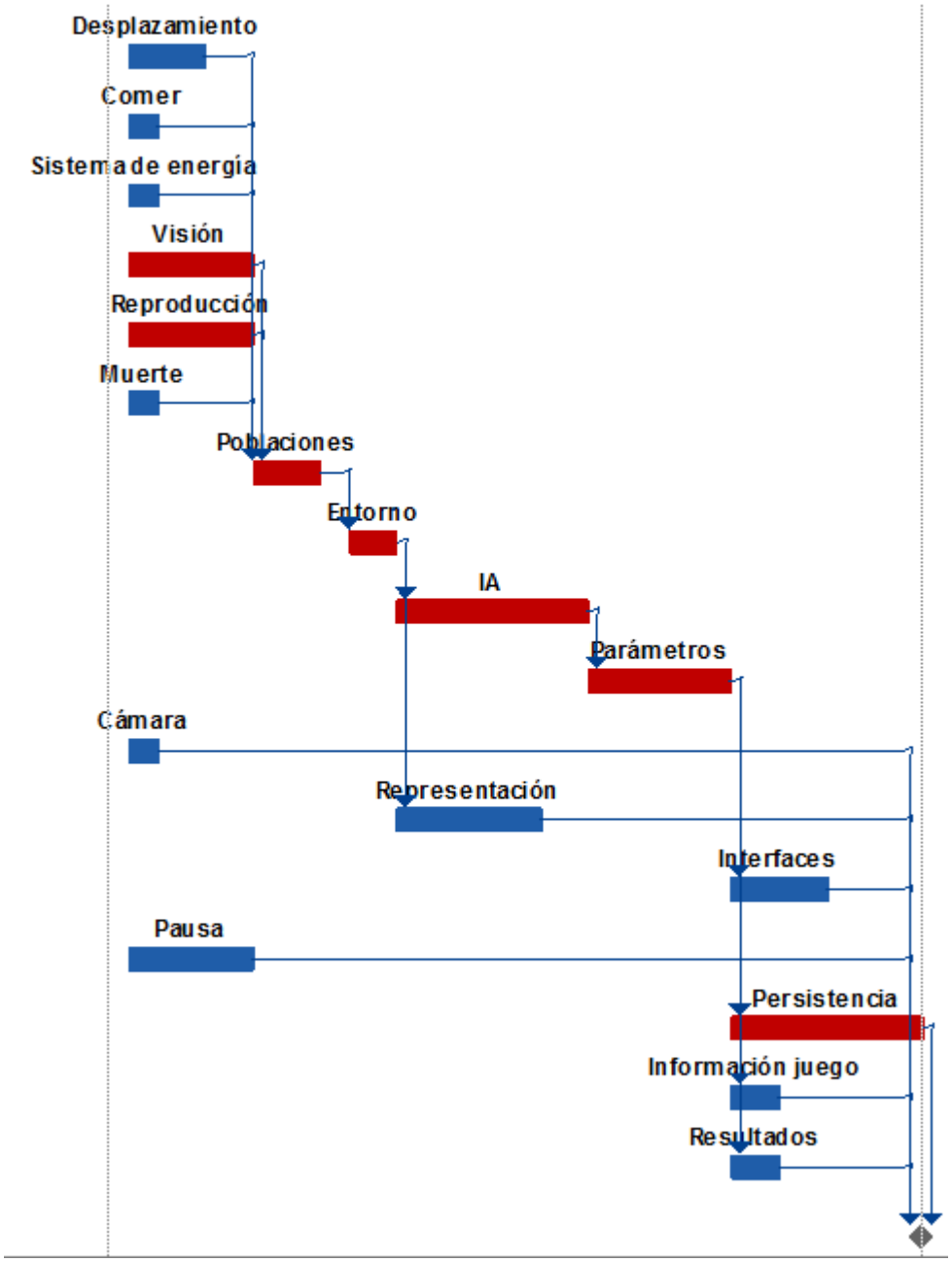


Ilustración 13. Diagrama de Gantt

6.3.2. Estimación de costes

Para realizar la estimación de costes habrá que tener en cuenta el personal, el software, el hardware y las instalaciones.

La mayoría del software utilizado es de licencia gratuita o con licencia para estudiantes así que no influirá en el coste.

Hardware tampoco ha sido necesario comprar por lo que no se incluirá en el coste.

Por otro lado, sí tendré en cuenta el gasto eléctrico del ordenador. El consumo del equipo es de 240 W por hora aproximadamente y la tarifa es de 0,126122 €/kWh.

En cuanto al personal, en este proyecto cumplo el rol de ingeniero de software. Tras realizar una búsqueda sobre esta ocupación en tusalario [15] e infojobs he comprobado que el salario oscila entre 20 y 80 euros por hora. Finalmente, he elegido 30 euros por hora por ser lo más frecuente.

Recurso	Coste	
Personal		
Ingeniero de software	300 horas * 30€/hora	9000€
Instalaciones		
Consumo eléctrico	300 horas * 0,24kW * 0,126122 €/kWh	9.08€
Total		9009,08€

Tabla 20. Estimación de costes

El coste total del desarrollo del proyecto sería de 9009,08 €.

6.4. Artefactos

En esta sección se especifican los artefactos utilizados para determinar los límites del sistema según los requisitos previamente recogidos.

Al documentar los requisitos, se distinguen tres tipos de perspectivas: de datos, de funcionamiento y de comportamiento. Cada perspectiva la he documentado por separado, utilizando lenguajes de modelado adecuados [16].

6.4.1. Casos de uso

Los casos de usos permiten definir la interacción entre el sistema y el exterior. Estos definen bajo la forma de acciones y reacciones el comportamiento del sistema desde el punto de vista del usuario (perspectiva de comportamiento).

Según los requisitos especificados, se pueden identificar las funcionalidades principales del sistema, que encapsularán diferentes casos de uso del sistema.

El diagrama general de casos de uso es el siguiente:

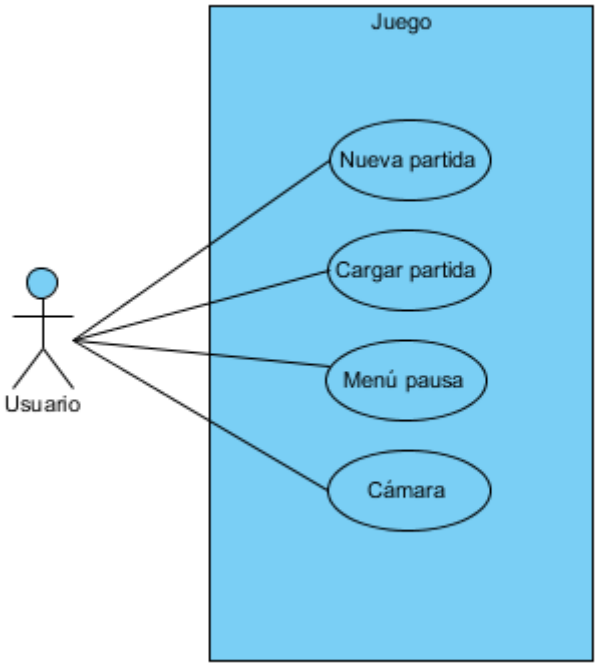


Ilustración 14. Diagrama de casos de uso

A continuación se detalla cada caso de uso:

- **Nueva partida**

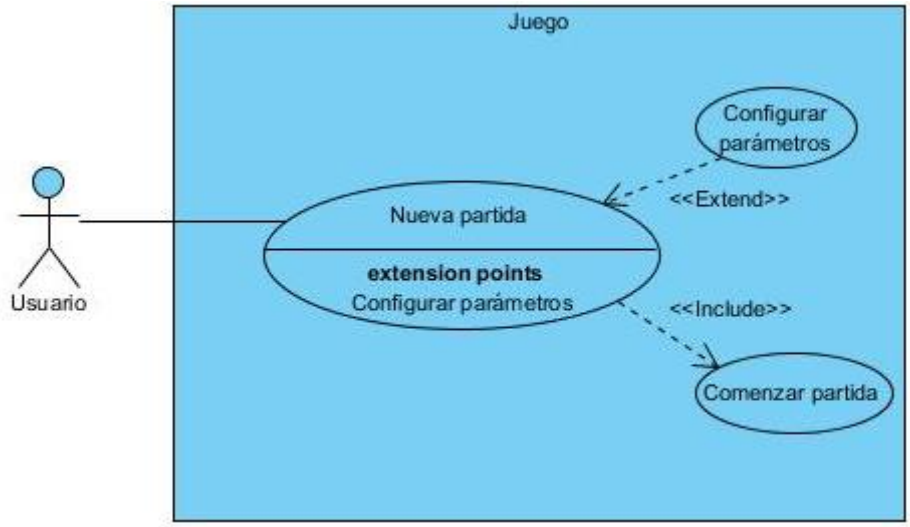


Ilustración 15. Caso de uso Nueva partida

- **Actores**: Usuario.
- **Condiciones de entrada**: -
- **Eventos**:
  1. El usuario selecciona crear una nueva partida.
  2. Se muestra la pantalla de parámetros.
  3. El usuario puede establecer los parámetros iniciales.
  4. El usuario selecciona empezar la partida.
  5. El sistema comienza la partida con los parámetros indicados.

- **Cargar partida**

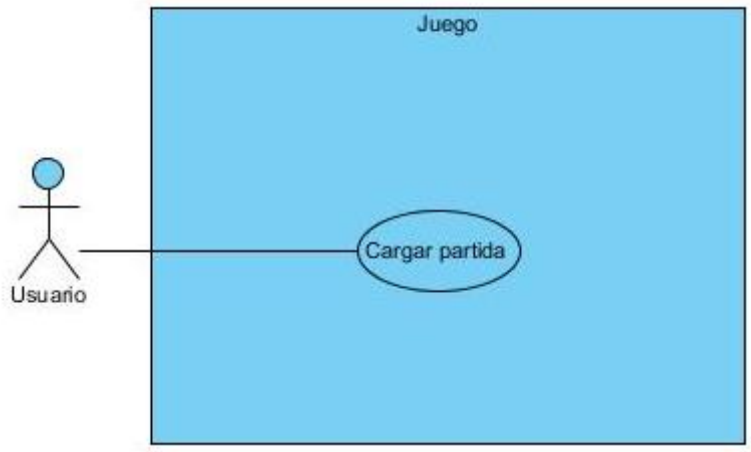


Ilustración 16. Caso de uso Cargar partida

- **Actores**: Usuario.
- **Condiciones de entrada**: -
- **Eventos**:
  1. El usuario selecciona cargar partida. (E1)
  2. El sistema comienza la partida con los parámetros indicados.
- **Excepciones**: E1: Si no existen datos guardados el sistema muestra un mensaje de error.

- **Menú pausa**

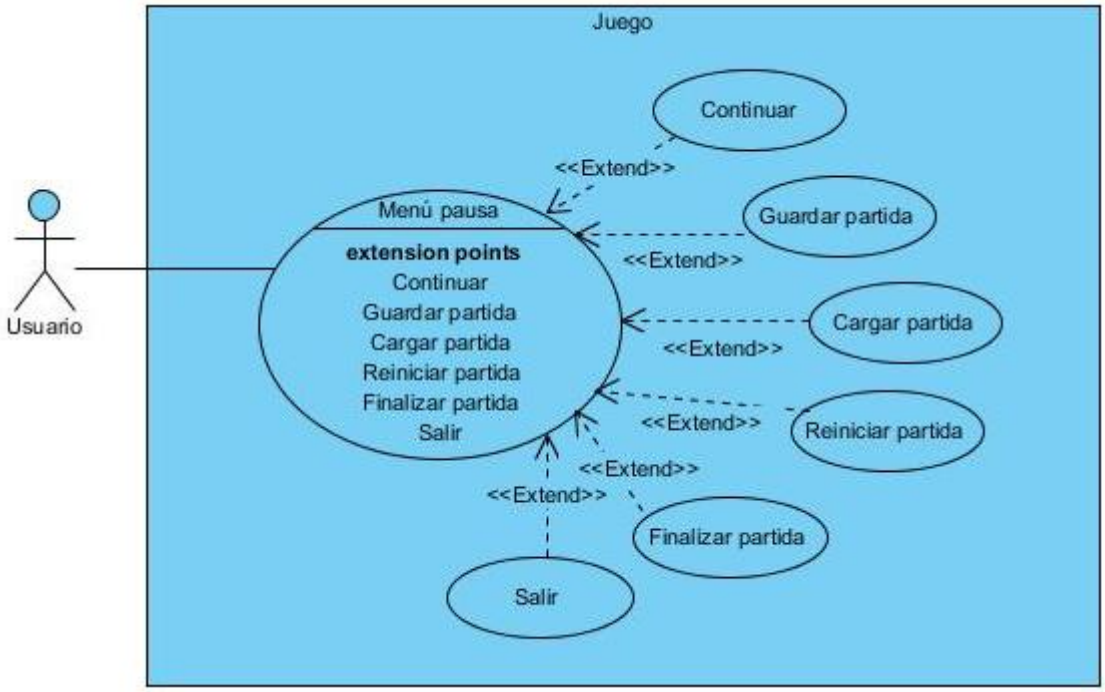


Ilustración 17. Caso de uso Menú pausa

- **Actores**: Usuario.
- **Condiciones de entrada**: partida en ejecución.
- **Eventos**:
  1. El usuario abre el menú pausa.
  2. El usuario puede seleccionar continuar la partida.
  3. El usuario puede guardar la partida
  4. El usuario puede cargar una partida existente. (E1)
  5. El usuario puede reinicializar la partida.
  6. El usuario puede finalizar la partida.
  7. El usuario puede salir al menú principal.
- **Excepciones**: E1: Si no existen datos guardados el sistema muestra un mensaje de error.

- **Cámara**

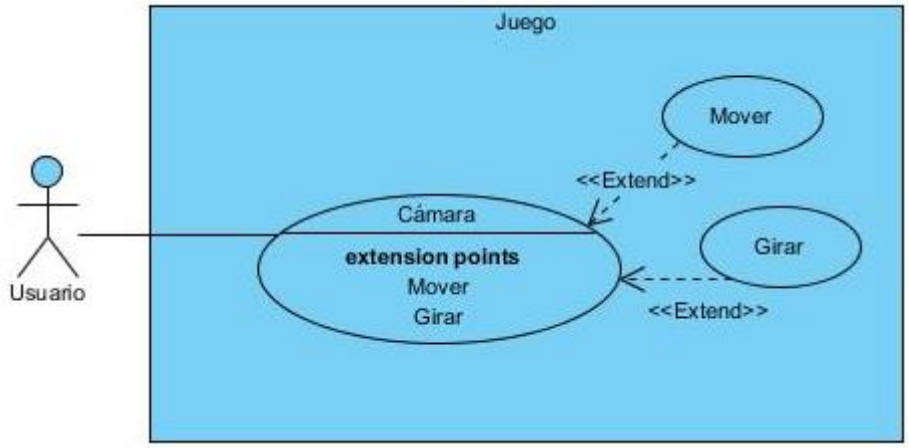


Ilustración 18. Caso de uso Cámara

- **Actores**: Usuario.
- **Condiciones de entrada**: partida en ejecución.
- **Eventos**:
  1. El usuario puede mover la posición de la cámara.
  2. El usuario puede girar la cámara.

6.4.2. Modelo de dominio

El modelo de dominio lo utilizo para documentar una visión estructural del sistema (perspectiva de datos). Esto permite ver qué conceptos se manejan en el contexto del problema y sus propiedades, para crear las clases de objetos.

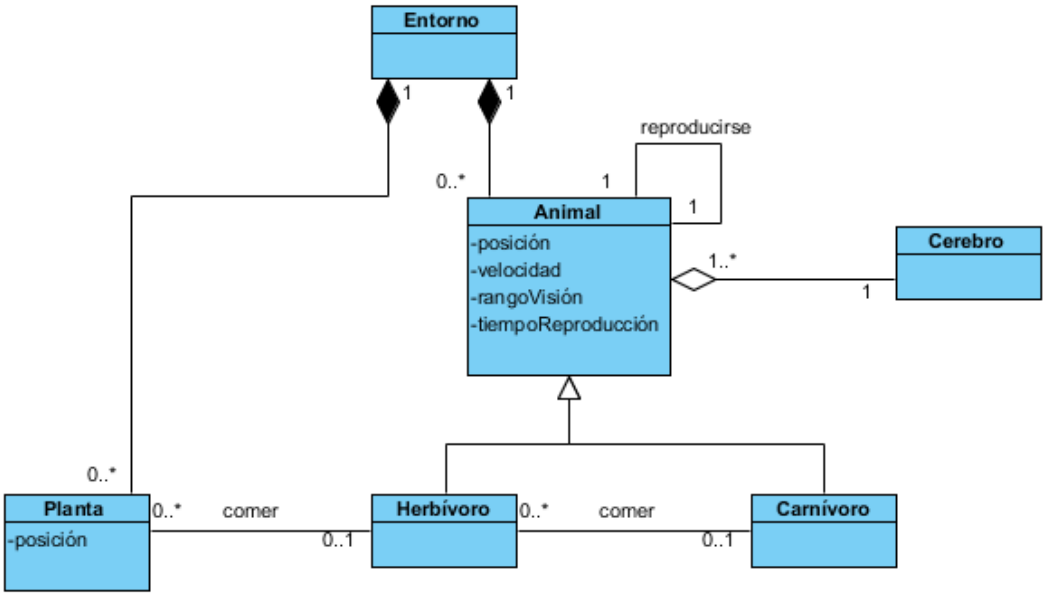


Ilustración 19. Modelo de dominio

6.5. Modelo de datos

Utilizo un modelo entidad-relación para representar las entidades relativas al almacenamiento y recuperación de la información.

A continuación se muestra el modelo E-R, que muestra las entidades, sus atributos y las relaciones entre ellas.

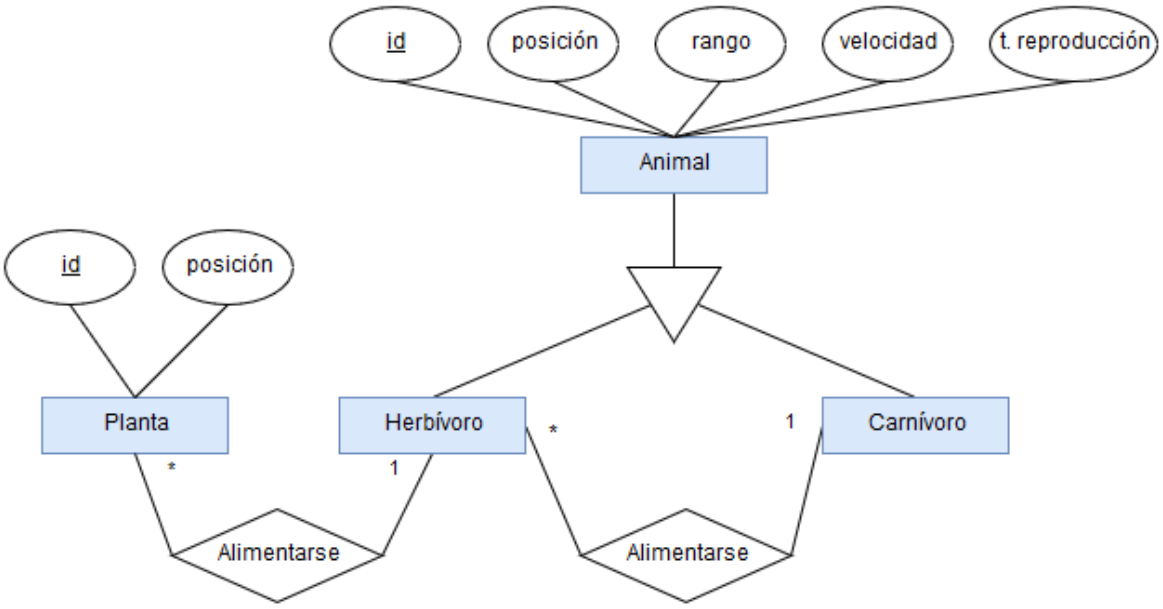


Ilustración 20. Modelo E-R

7. DISEÑO

7.1. Artefactos

7.1.1. Diagrama de clases

En el análisis, mediante el modelo de dominio, he definido un conjunto de clases de análisis que describen los elementos del dominio del problema.

En el diseño defino un conjunto de clases de diseño refinando las clases del análisis y produciendo otras nuevas que permitan el desarrollo de la solución.

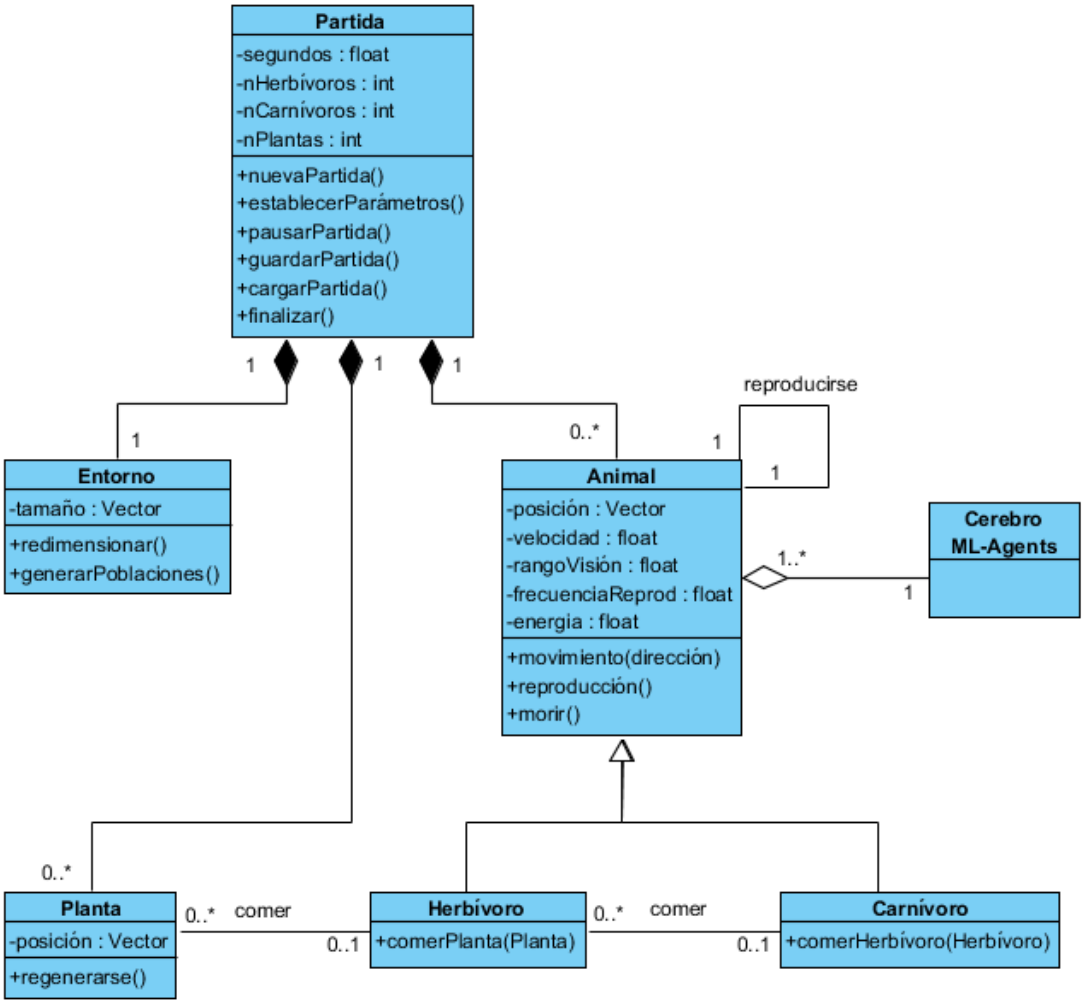


Ilustración 21. Diagrama de clases

7.1.2. Diagramas de secuencia

Los diagramas de secuencia permiten ver la serie de interacciones que un usuario debe seguir para alcanzar un objetivo.

He realizado diagramas de secuencia para los casos de uso que implican una serie de pasos y permiten a los usuarios alcanzar objetivos.

- Diagrama de secuencia del sistema de "**Nueva partida**":

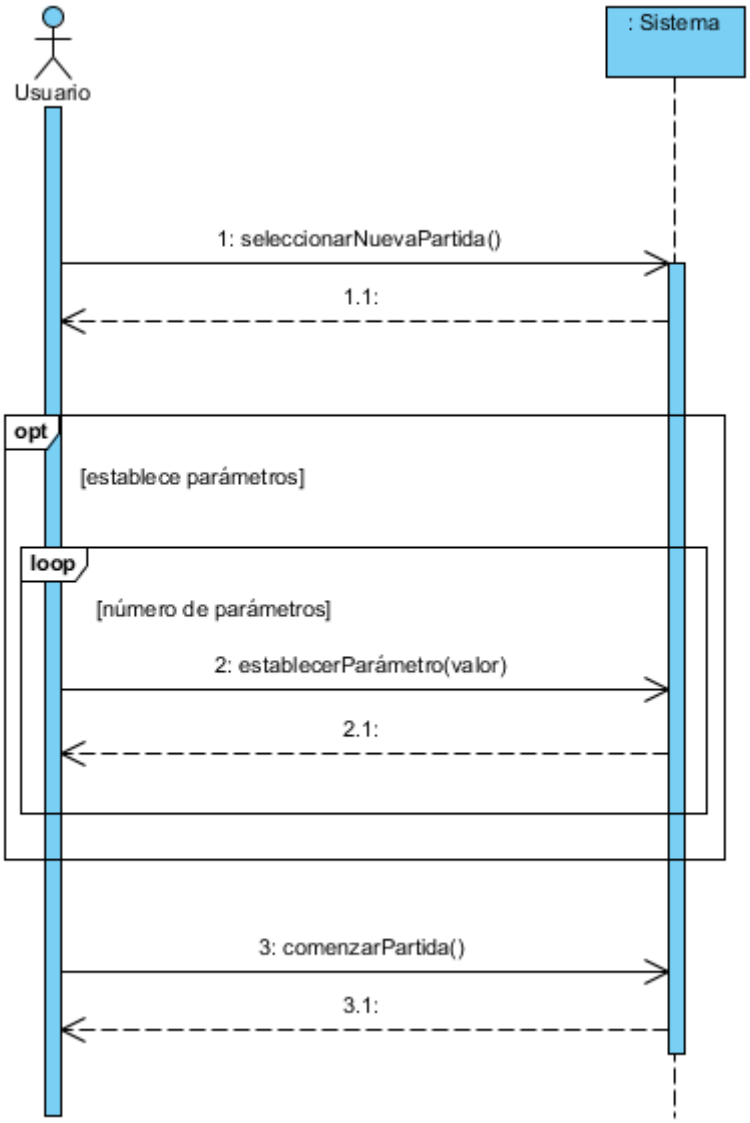


Ilustración 22. Diagrama de secuencia de Nueva Partida

- Diagrama de secuencia del sistema de “**Guardar partida**”:

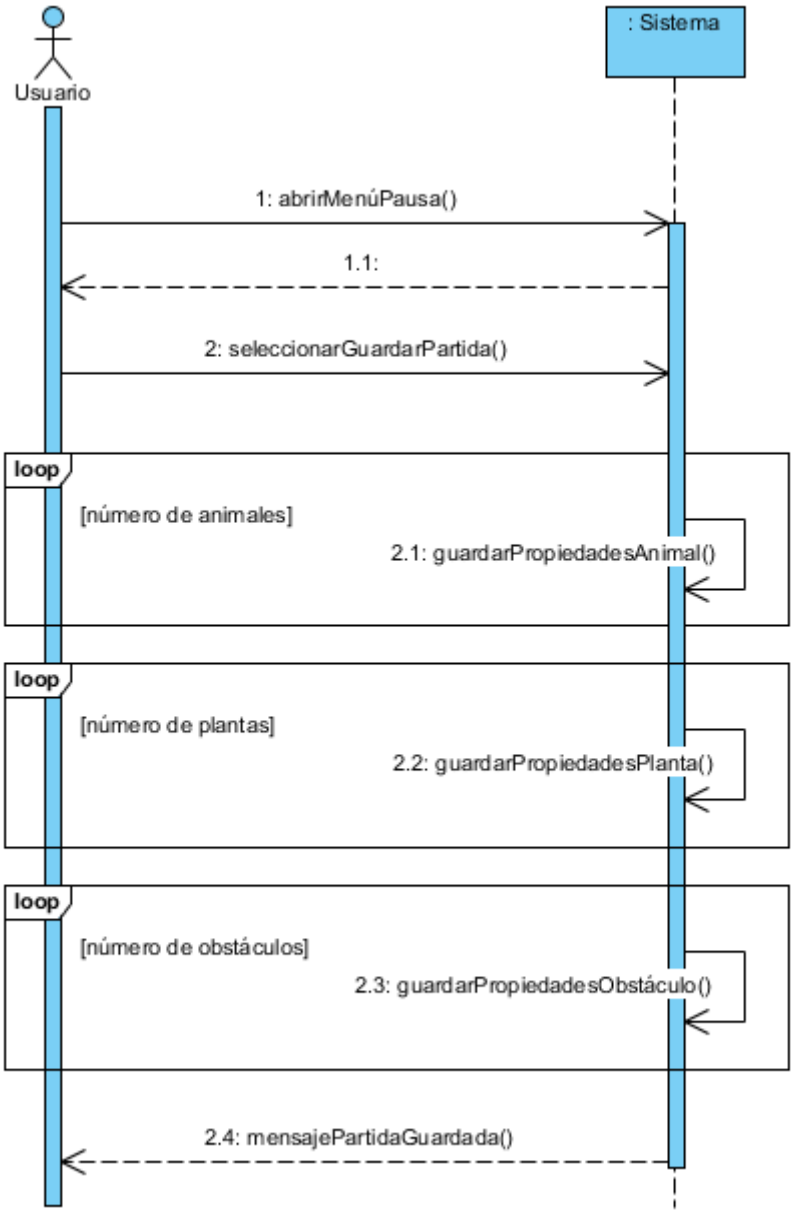


Ilustración 23. Diagrama de secuencia de Guardar Partida

- Diagrama de secuencia del sistema de “**Cargar partida**”:

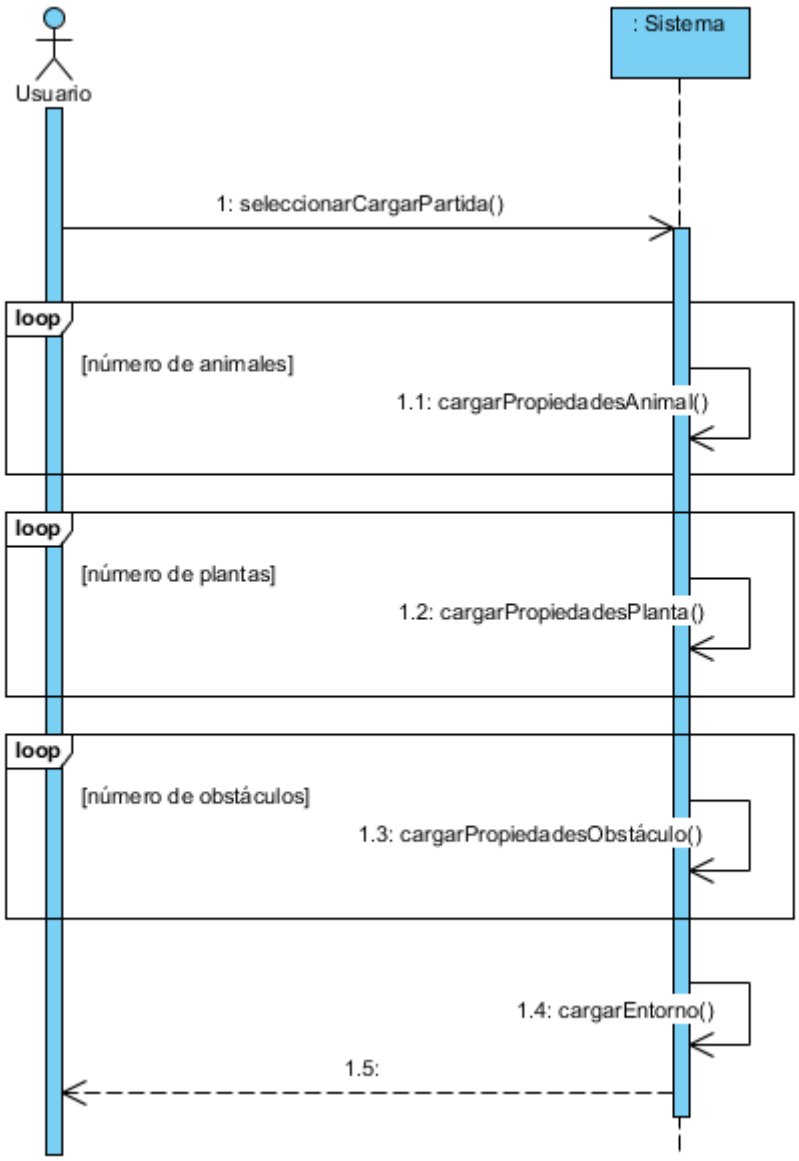


Ilustración 24. Diagrama de secuencia de Cargar Partida

7.2. Mecánica de juego

La mecánica de juego es el conjunto de reglas que gobiernan la interacción entre las entidades de un juego [17].

7.2.1. Objetivos del jugador

El objetivo del jugador será llegar a conseguir establecer unos parámetros iniciales que consiga un estado de equilibrio entre las poblaciones de conejos, lobos y plantas.

El número de lobos deberá ser el adecuado de forma que no se extingan todos los conejos y les dé tiempo a multiplicarse. Por otro lado, el número de plantas deberá ser el apropiado para que los conejos no mueran de hambre. Lo mismo pasará con el número de conejos.

Pero no solo el tamaño de las poblaciones se podrá establecer. También se podrán indicar características de los animales, como la velocidad de movimiento y el rango de visión.

Además, se podrá seleccionar el modelo entrenado que utilizarán los agentes.

7.2.2. Métricas de éxito

El éxito se medirá según el equilibrio entre las poblaciones y se mostrará al jugador en la pantalla de resultados.

Para mostrar la pantalla de resultados tiene que finalizar la partida. El juego finaliza cuando alguna población se extingue o si se selecciona finalizar.

En la pantalla de resultados se mostrarán gráficas que muestren el cambio en el tamaño de las poblaciones y también se mostrará el tiempo de juego.

7.2.3. Habilidades del jugador

Tras introducir los parámetros y comenzar la partida, el jugador podrá mover la cámara libremente por el entorno mediante el teclado y el ratón.

En cualquier momento puede pausar la partida para acceder al menú pausa. Desde aquí podrá reanudar la partida, guardarla, cargar una previamente guardada, reiniciarla, finalizarla para pasar a la pantalla de resultados y salir al menú principal.

También se podrá cargar una partida desde el menú principal.

7.3. Flowboard

Los flowboard se suelen utilizar para describir las transiciones en la jugabilidad de un videojuego [18].

A continuación se puede observar el flowboard de la aplicación:

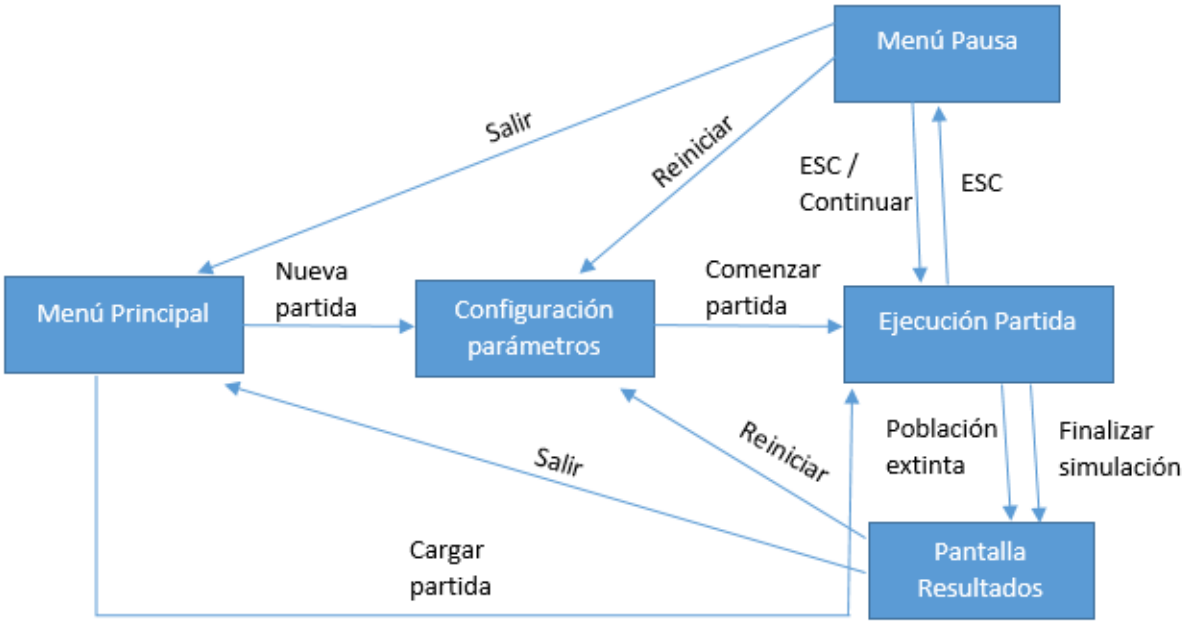


Ilustración 25. Flowboard

7.4. Diseño del mundo

El entorno del juego será un bosque. En este habrá árboles, hierba y plantas.

Los herbívoros se alimentarán de las plantas mientras que los carnívoros se alimentarán de los animales herbívoros. Las plantas tendrán un tiempo de regeneración tras su consumición.

Los animales tendrán que moverse por el entorno evitando los obstáculos del entorno, que serán los árboles.

La posición inicial de todos estos elementos del entorno será aleatoria.

Además, el entorno será redimensionable, es decir, su tamaño dependerá de los parámetros iniciales.

A continuación se pueden ver un mapa del entorno con sus diferentes elementos:

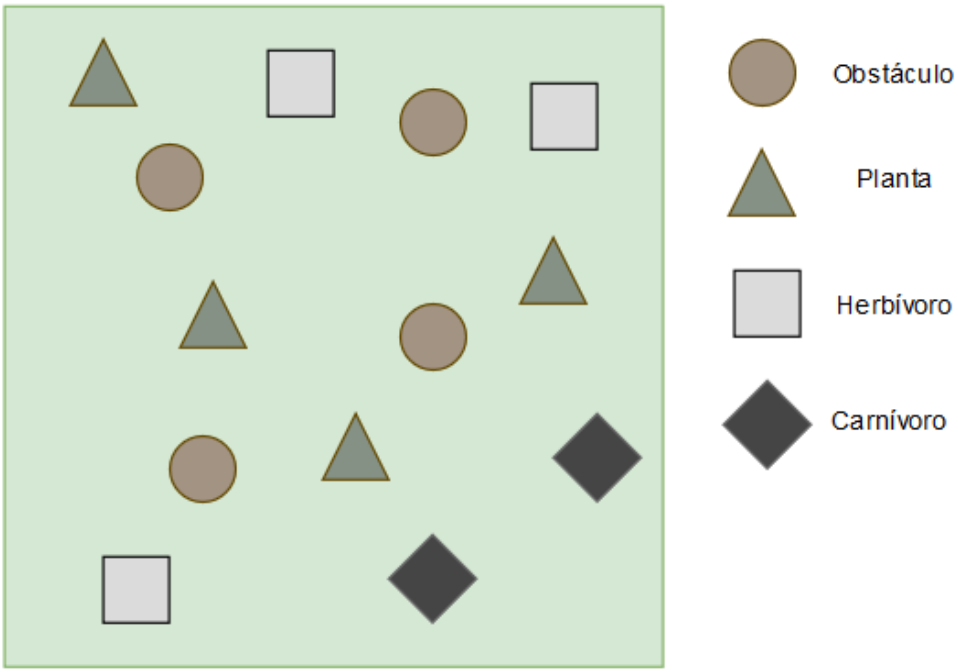


Ilustración 26. Mapa del entorno

7.5. Diseño de personajes

Para representar los animales herbívoros y carnívoros utilizaré conejos y lobos. A continuación describo las características y objetivos de cada uno:

7.5.1. Conejos

- Características y habilidades:
  - Movimiento por el entorno.
  - Comer plantas
  - Reproducirse transmitiendo características
  - Muerte si se queda sin energía o es comido por un lobo.
- Objetivos: alimentarse de plantas, reproducirse y huir de los lobos.

7.5.2. Lobos

- Características y habilidades:
  - Movimiento por el entorno.

    - Comer conejos
    - Reproducirse transmitiendo características
    - Muerte si se queda sin energía.
- Objetivos: alimentarse de conejos y reproducirse.

8. IMPLEMENTACIÓN

En este apartado explico los aspectos principales de ML-Agents, detallo el proceso seguido para conseguir un comportamiento inteligente y la implementación de la aplicación en la que se utilizará.

Utilizaré términos específicos de Unity y ML-Agents por lo que he incluido un glosario con algunos de los términos más importantes como primer anexo de esta memoria.

8.1. Características ML-Agents

Para poder trabajar con ML-Agents primero tuve que realizar un estudio sobre su funcionamiento.

A continuación explico los aspectos principales de ML-Agents para posteriormente detallar su implementación.

8.1.1. Entrenamiento de comportamientos

Para poder obtener comportamientos inteligentes utilizando ML-Agents, primero es necesario definir tres entidades en el entorno [19]:

- **Observaciones**: todo lo que el agente percibe del entorno. Las observaciones pueden ser numéricas y/o visuales. Las observaciones numéricas miden los atributos del entorno desde el punto de vista del agente. Estas pueden ser discretas o continuas dependiendo de la complejidad del juego y del agente. Para los entornos más complejos, un agente requerirá varias observaciones numéricas continuas, mientras que para entornos simples una observación discreta será suficiente. Las observaciones visuales, por otro lado, son imágenes generadas por cámaras conectadas al agente y representan lo que el agente está viendo en ese momento. Es importante no confundir la observación de un agente con el **estado** del entorno. El estado del entorno representa información sobre toda la escena. Sin embargo, la

observación de los agentes solo contiene información que el agente conoce y suele ser un subconjunto del estado del entorno.

- **Acciones**: operaciones que puede realizar el agente. De forma similar a las observaciones, las acciones pueden ser continuas o discretas dependiendo de la complejidad del entorno y del agente.
- **Señales de recompensa**: valor escalar que indica cómo de bien lo está haciendo el agente. La señal de recompensa no necesita proporcionarse en todo momento, sino solo cuando el agente realiza una acción que se considera buena o mala. Hay que tener en cuenta que la señal de recompensa es la forma en que los objetivos de la tarea se comunican al agente, por lo que deben configurarse de manera que la maximización de la recompensa genere el comportamiento óptimo deseado.

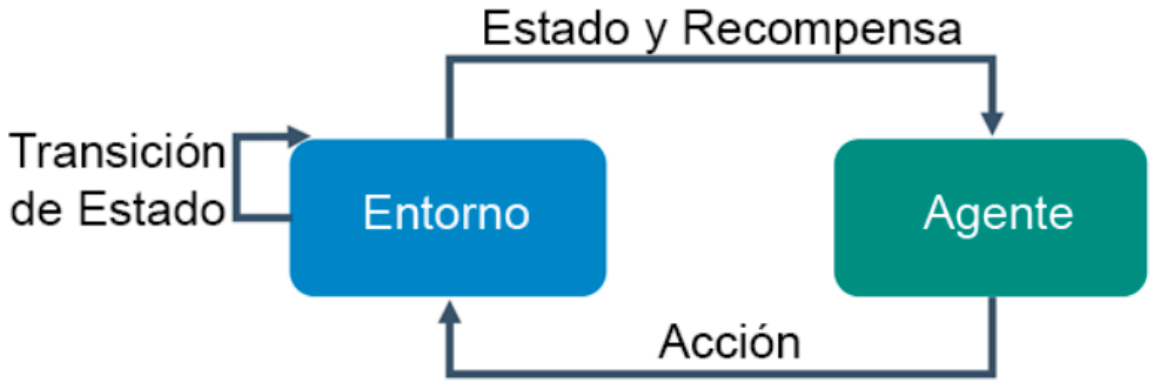


Ilustración 27. Ciclo del aprendizaje reforzado

Tras definir estas tres entidades (los componentes básicos de una tarea de aprendizaje reforzado), se puede entrenar el comportamiento del agente. Esto se logra simulando el entorno durante muchas pruebas en las que el agente, con el tiempo, aprende cuál es la acción óptima que debe tomar para cada observación maximizando su recompensa futura.

En la terminología de aprendizaje reforzado, el comportamiento que se aprende se denomina **política**, que es básicamente un mapeo (óptimo) de observaciones a acciones. El proceso de aprendizaje de una política mediante la ejecución de simulaciones se denomina **fase de entrenamiento**, mientras que el juego con un

agente que utiliza su política aprendida se denomina **fase de inferencia**.

ML-Agents proporciona las herramientas que permiten utilizar Unity como motor de simulación para aprender políticas en un entorno de entrenamiento.

8.1.2. Componentes de ML-Agents

ML-Agents contiene tres componentes de alto nivel:

- **Entorno de aprendizaje**: contiene la escena de Unity y los personajes del juego.
- **API de Python**: contiene los algoritmos de aprendizaje automático que se utilizan para el entrenamiento. A diferencia del entorno de aprendizaje, la API de Python no forma parte de Unity, sino que se comunica a través del comunicador externo.
- **Comunicador externo**: conecta el entorno de aprendizaje con la API de Python. Forma parte del entorno de aprendizaje.

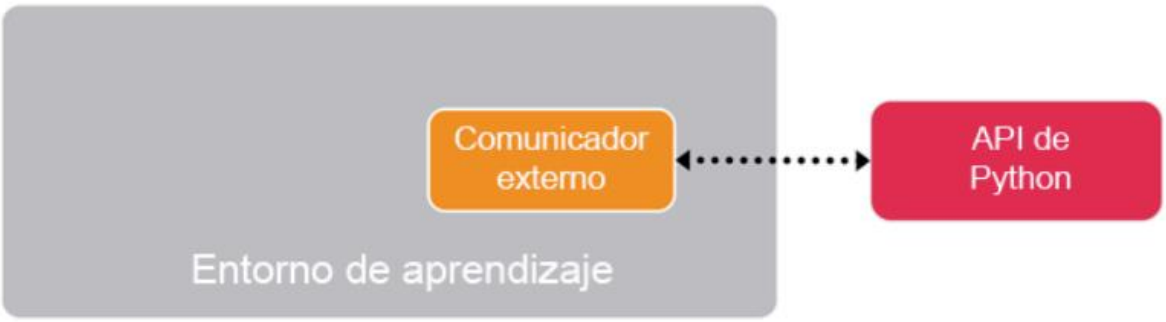


Ilustración 28. Componentes de ML-Agents

8.1.3. Entorno de aprendizaje

El entorno de aprendizaje contiene tres tipos de objetos:

- **Agente**: controla la generación de **observaciones**, la realización de **acciones** y la asignación de una **recompensa** (positiva o negativa) cuando corresponda. Cada agente está vinculado a un cerebro.
- **Cerebro**: encapsula la lógica de toma de decisiones para el agente. Es decir, el cerebro es el que determina qué acciones debe tomar el

agente en cada instancia tras recibir sus observaciones y recompensas.

- **Academia**: organiza el proceso de observación y toma de decisiones. La academia se asegura de que todos los agentes y cerebros estén sincronizados, además de controlar la configuración de todo el entorno. Dentro de la academia, se pueden especificar varios parámetros del entorno, como la calidad de representación y la velocidad a la que se ejecuta el entorno. El comunicador externo forma parte de la academia. Cada entorno contiene una sola academia que define el alcance del entorno, en términos de:

  - Configuración del motor: la velocidad y la calidad de representación del motor del juego en los modos de entrenamiento e inferencia.
  - Longitud del episodio global: cuánto durará el episodio de entrenamiento.

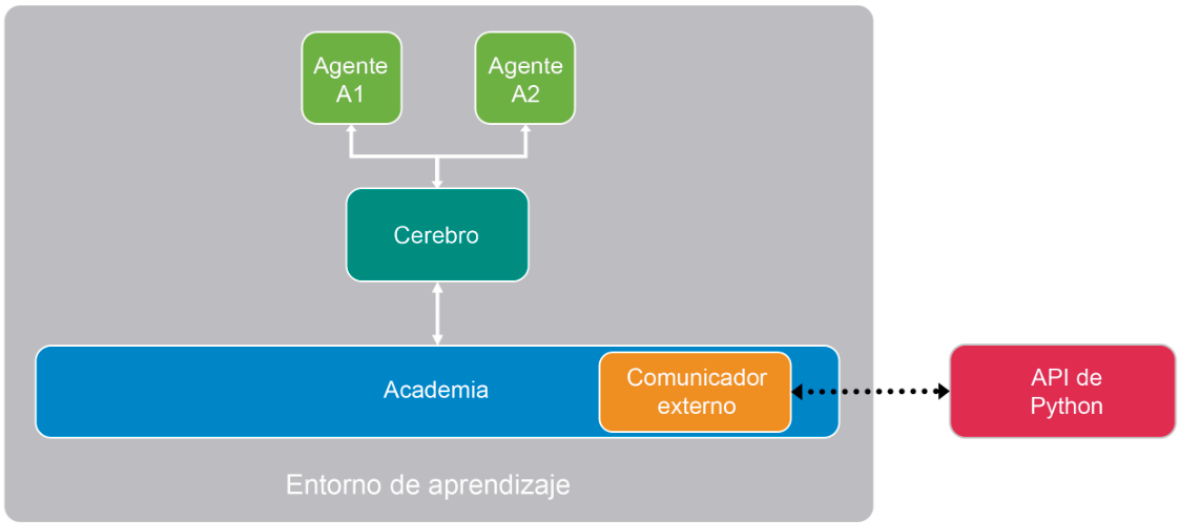


Ilustración 29. Ejemplo de entorno en ML-Agents y sus componentes

Cada entorno de aprendizaje siempre tendrá una academia global y un agente para cada personaje de la escena. Si bien cada agente debe estar vinculado a un solo cerebro, los agentes que tienen observaciones y acciones similares pueden estar vinculados al mismo cerebro. Esto no significa que tengan en cada instancia los mismos valores de observaciones y acciones. El cerebro define el espacio de

observaciones y acciones posibles y los agentes conectados a él tienen sus propios valores.

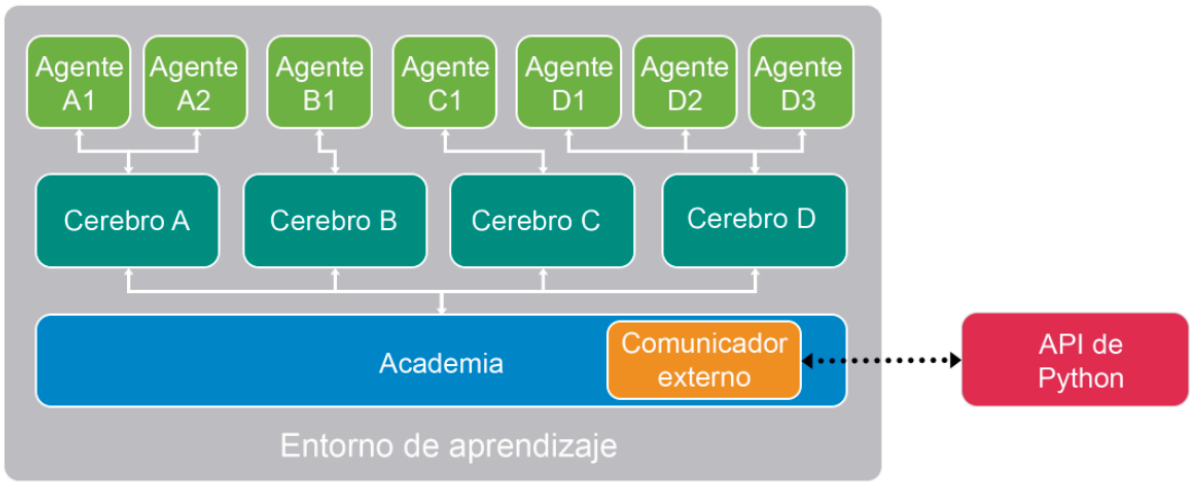


Ilustración 30. Ejemplo de entorno con múltiples agentes y cerebros

En la práctica existen cuatro tipos diferentes de cerebros que permiten diferentes escenarios de entrenamiento e inferencia:

- **Externo**: las decisiones se toman utilizando la API de Python. Las observaciones y recompensas recopiladas por el cerebro se envían a la API de Python a través del comunicador externo. Tras esto, la API de Python devuelve la acción correspondiente que debe tomar el agente.
- **Interno**: donde las decisiones se toman usando un **modelo entrenado** integrado en el proyecto a través de TensorFlow. El modelo de TensorFlow representa una política aprendida y el cerebro usa directamente este modelo para determinar la acción de cada agente.
- **Jugador**: donde las decisiones se toman usando la entrada de un dispositivo. Es decir, un usuario humano controla al agente por lo que las observaciones y recompensas recopiladas por el cerebro no se utilizan para controlarlo. Este tipo de cerebro solo se suele utilizar para realizar pruebas.
- **Heurístico**: donde las decisiones se toman usando un comportamiento prediseñado. Esto se asemeja a cómo la mayoría de los comportamientos de los personajes se definen actualmente y puede ser útil para depurar o

comparar cómo un agente con reglas prediseñadas se compara con un agente cuyo comportamiento ha sido entrenado.

8.1.4. Entrenamiento e inferencia

Para la fase de entrenamiento el cerebro se debe establecer a tipo externo mientras que en la fase de inferencia debe establecerse a interno.

Durante el **entrenamiento**, todos los agentes (con cerebro externo) de la escena envían sus observaciones a la API de Python a través del comunicador externo. La API de Python procesa estas observaciones y devuelve las acciones para cada agente. Estas acciones son principalmente exploratorias para ayudar a la API de Python a aprender la mejor política para cada agente. Una vez concluido el entrenamiento, se puede exportar la política aprendida para cada agente. Dado que las implementaciones se basan en TensorFlow, la política aprendida es simplemente un archivo de modelo de TensorFlow.

Durante la fase de **inferencia**, se debe cambiar el tipo de cerebro a interno e incluir el modelo de TensorFlow generado a partir de la fase de entrenamiento. En esta fase, los agentes aún continúan generando sus observaciones, pero en lugar de enviarlos a la API de Python, se pasarán a su modelo para generar la acción óptima para cada agente en cada instancia.

8.1.5. Escenarios posibles

Utilizando los agentes de ML-Agents se pueden crear diferentes tipos de escenarios de entrenamiento, dependiendo de la forma en que se conecten los agentes, cerebros y recompensas:

- **Agente único**: un agente vinculado a un solo cerebro, con su propia señal de recompensa. Esta es la forma tradicional de entrenar a un agente. Un ejemplo es cualquier juego para un solo jugador.

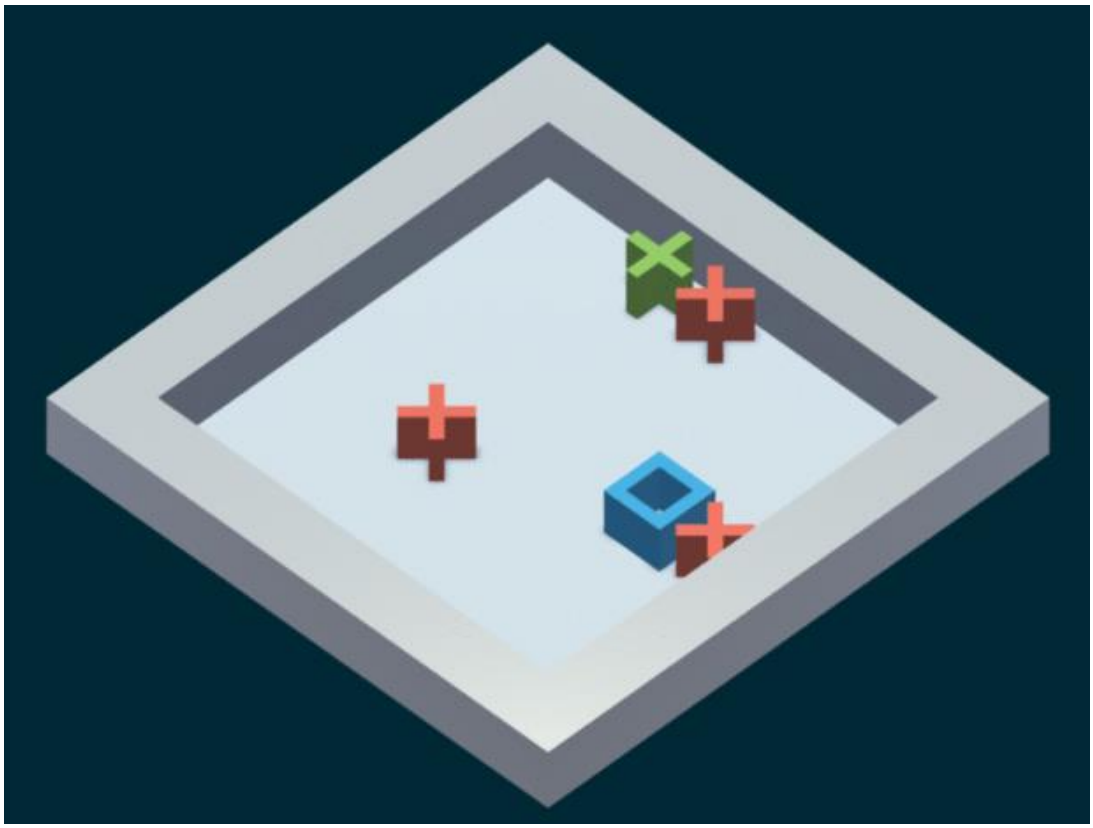


Ilustración 31. Ejemplo de agente único: un agente (objeto azul) vinculado a un solo cerebro tiene el objetivo de llegar a una casilla (en verde) sin pasar por las marcadas en rojo

- **Agente único simultáneo**: varios agentes independientes con señales de recompensa independientes vinculadas a un solo cerebro. Una versión paralela del escenario de entrenamiento tradicional, que puede acelerar y estabilizar el proceso de entrenamiento. Este escenario es útil cuando hay múltiples versiones del mismo personaje en un entorno que debería aprender comportamientos similares.

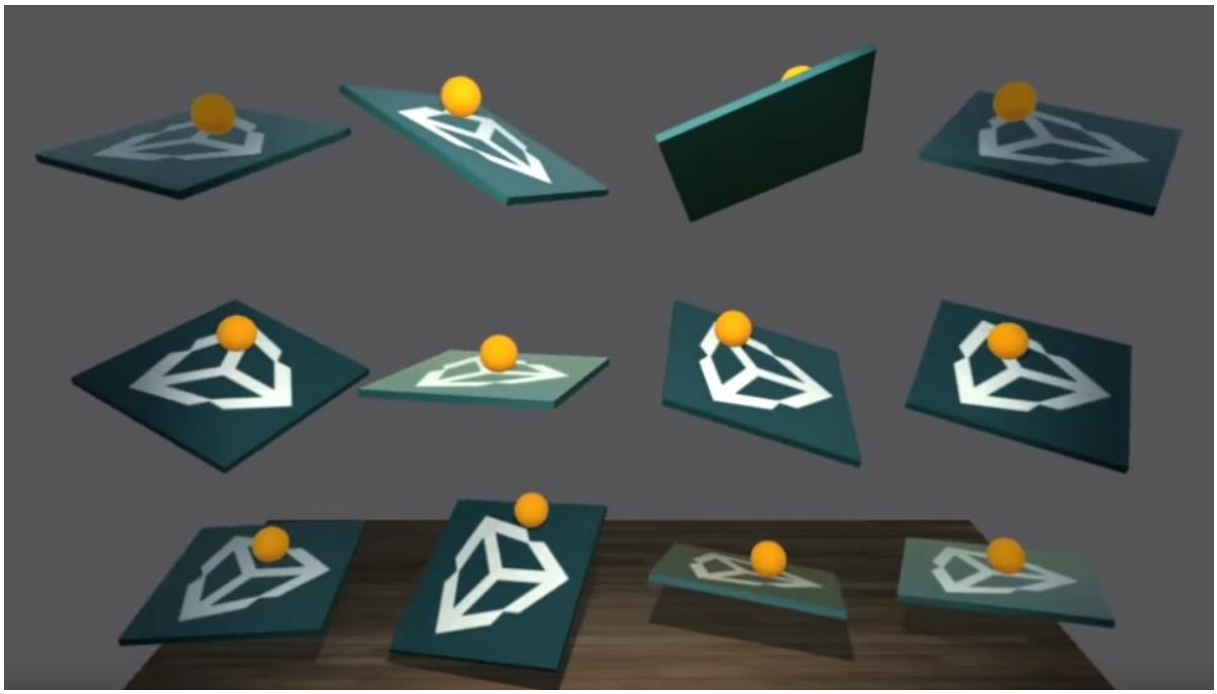


Ilustración 32. Ejemplo de agente único simultáneo: agentes (plataformas) vinculados a un solo cerebro tienen el objetivo de evitar que la bola se caiga

- **Auto-adversario**: dos agentes que interactúan con señales de recompensa inversa vinculadas a un solo cerebro. En los juegos de dos jugadores, un escenario “auto-adversario” puede permitir que un agente mejore sus habilidades de forma continua al ser él mismo su propio oponente. Esta fue la estrategia empleada en el entrenamiento de AlphaGo y, más recientemente, utilizada por OpenAI para entrenar a un agente de Dota 2 [20].

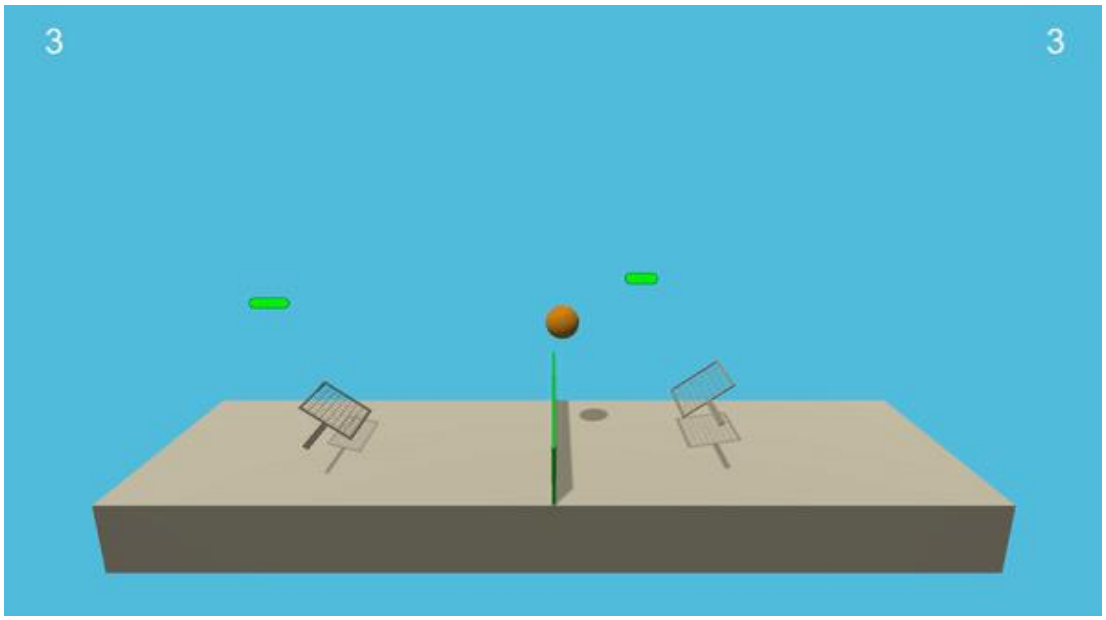


Ilustración 33. Ejemplo de escenario auto-adversario: juego de tenis con dos agentes con señales de recompensa inversa vinculadas a un solo cerebro

- **Multi-Agente cooperativo**: varios agentes que interactúan con una señal de recompensa compartida vinculada a uno o varios cerebros diferentes. En este escenario, todos los agentes deben trabajar juntos para lograr una tarea que no se puede hacer individualmente. Un ejemplo sería un entorno en el que cada agente tiene acceso a información parcial que debe compartirse para realizar una tarea.
- **Multi-agente competitivo**: varios agentes que interactúan con señales de recompensa inversa vinculadas a uno o varios cerebros diferentes. En este escenario, los agentes deben competir entre sí para ganar una competición u obtener un conjunto limitado de recursos. Un ejemplo serían los deportes de equipo.
- **Ecosistema**: varios agentes que interactúan con señales de recompensa independientes vinculadas a uno o varios cerebros diferentes. Un ejemplo de este escenario podría ser una simulación de conducción autónoma dentro de un entorno urbano. Este es el tipo de escenario más adecuado para este proyecto, ya que el entorno de entrenamiento estará formado por poblaciones de animales con objetivos diferentes.

8.1.6. Proximal Policy Optimization

ML-Agents utiliza una técnica de aprendizaje reforzado llamada Proximal Policy Optimization (PPO). PPO emplea una red neuronal con el objetivo de aproximarse a la función ideal que mapea las observaciones de un agente con la mejor acción que este puede realizar en un estado determinado. El algoritmo PPO de ML-Agents está implementado en TensorFlow y se ejecuta en un proceso de Python separado (se comunica por *socket* con la aplicación de Unity en ejecución) [21].

PPO ha sido desarrollado por OpenAI, una compañía dedicada a la investigación en inteligencia artificial. Actualmente es el algoritmo de aprendizaje reforzado utilizado por OpenAI debido a su buen rendimiento y facilidad de uso en comparación con otras técnicas existentes [22].

PPO es un método de gradiente de política para el aprendizaje reforzado, que alterna entre el muestreo de datos a través de la interacción con el entorno y la optimización de una función objetivo utilizando descenso de gradiente estocástico [23]. Aunque el nombre para referirse a esta técnica sea descenso de gradiente estocástico (SGD, Stochastic Gradient Descent), puede utilizarse ascenso o descenso de gradiente. PPO, en concreto, utiliza ascenso ya que su objetivo es maximizar la recompensa.

Los métodos de gradiente de política son un tipo de técnicas de aprendizaje reforzado que se basan en la optimización de políticas parametrizadas respecto al rendimiento esperado (recompensa acumulada a largo plazo) por descenso de gradiente. Estos métodos evitan muchos de los problemas que han estado perjudicando los enfoques tradicionales de aprendizaje reforzado, como los resultantes de información de estado incierta y la complejidad que surge de estados y acciones continuas [24].

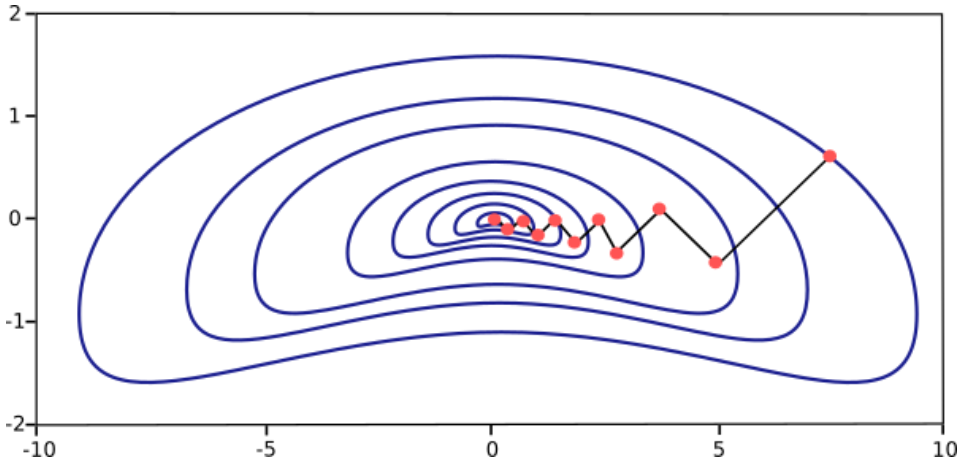


Ilustración 34. Descenso de gradiente

Los métodos de gradiente de política son fundamentales para avances recientes en el uso de redes neuronales para el control de videojuegos o locomoción 3D [25]. Sin embargo, obtener buenos resultados a través de estos puede ser complicado porque son susceptibles a la elección del tamaño de cada paso (cuánto descender en cada paso): demasiado pequeño y el progreso es demasiado lento; demasiado grande y la señal se ve saturada por el ruido, o se podrían ver grandes caídas en el rendimiento. También suelen tener una eficiencia muy baja, necesitando millones (o miles de millones) de pasos para aprender tareas simples. Además, es necesario un ajuste de hiperparámetros previo.

Mediante PPO es posible obtener buenos resultados al lograr un equilibrio entre la facilidad de implementación, la complejidad de la muestra y la facilidad de ajuste. Esto lo consigue al calcular, en cada paso, una actualización intentando minimizar la función de coste al tiempo que garantiza que la desviación respecto a la política previa sea relativamente pequeña.

8.1.7. Instalación

Para la instalación y uso de ML-Agents es necesario instalar Unity, clonar el repositorio de ML-Agents e instalar Python con dependencias adicionales. Para la versión 0.2 también se requiere CUDA y la biblioteca cuDNN de Nvidia, que utiliza ML-Agents para realizar el entrenamiento mediante GPU [26].

8.1.8. Creación de un entorno de aprendizaje en Unity

En ML-Agents, un entorno de aprendizaje consiste en una aplicación creada mediante Unity que se puede utilizar para entrenar a los agentes mediante aprendizaje reforzado [27].

El uso de ML-Agents en un proyecto de Unity implica los siguientes pasos:

1. Crear un entorno para los agentes. Un entorno puede abarcar desde una simple simulación física que contiene unos pocos objetos hasta un juego o ecosistema completo.
2. Implementar una subclase de *Academy* (una clase abstracta que contiene la funcionalidad de la academia de ML-Agents) y agregarla a un *GameObject* (objeto en la escena de Unity) del entorno. Este *GameObject* servirá como padre en la jerarquía de la escena para cualquier *GameObject* de *Brain* (clase con funcionalidad de cerebro). La subclase de *Academy* puede implementar algunos métodos opcionales para actualizar la escena independientemente de cualquier agente. Por ejemplo, se puede agregar, mover o eliminar agentes y otras entidades en el entorno.
3. Agregar uno o más *GameObjects* cerebro a la escena como hijos de la academia en la jerarquía de la escena.

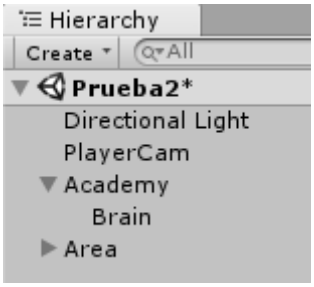


Ilustración 35. Ventana de la jerarquía de la escena. Se pueden observar los *GameObjects de la academia y de un cerebro*

4. Implementar las subclases de *Agent* (clase agente). En una subclase de *Agent* se define el código que un agente utiliza para observar su entorno, llevar a cabo las acciones asignadas y calcular las recompensas utilizadas para el entrenamiento. También se pueden implementar métodos opcionales para reiniciar el agente cuando finaliza o falla su tarea.

5. Agregar las subclases de *Agent* a *GameObjects* apropiados, generalmente, el objeto en la escena que representa el agente en la simulación. A cada *GameObject* agente se le debe asignar un *GameObject* cerebro.
6. Configurar el tipo de cerebro a externo y ejecutar el proceso de entrenamiento.

8.1.9. Características de la academia

El objeto academia organiza los agentes y sus procesos de toma de decisiones. Se necesita solo un objeto academia en una escena [28].

Tras crear una subclase de *Academy* (ya que la clase base es abstracta) se pueden implementar los siguientes métodos (opcionales):

- *InitializeAcademy()*: prepara el entorno la primera vez que se inicia.
- *AcademyReset()*: prepara el entorno y los agentes para el siguiente episodio del entrenamiento. Esta función se utiliza para colocar e inicializar entidades en la escena según sea necesario.
- *AcademyStep()*: prepara el entorno para el siguiente paso de la simulación. La clase base *Academy* llama a esta función antes de llamar a cualquier método *AgentAction()* para el paso actual. Se puede usar esta función para actualizar otros objetos en la escena antes de que los agentes realicen sus acciones, teniendo en cuenta que los agentes ya han recopilado sus observaciones y elegido una acción antes de que la academia invoque este método.

La clase base *Academy* también define varias propiedades importantes que se pueden establecer en el inspector de Unity. Para el entrenamiento, la más importante de estas propiedades es *max steps* (número de pasos máximo), que determina cuánto dura cada episodio del entrenamiento. Una vez que el contador de pasos de la academia alcanza este valor, llama a la función *AcademyReset()* para iniciar el siguiente episodio.

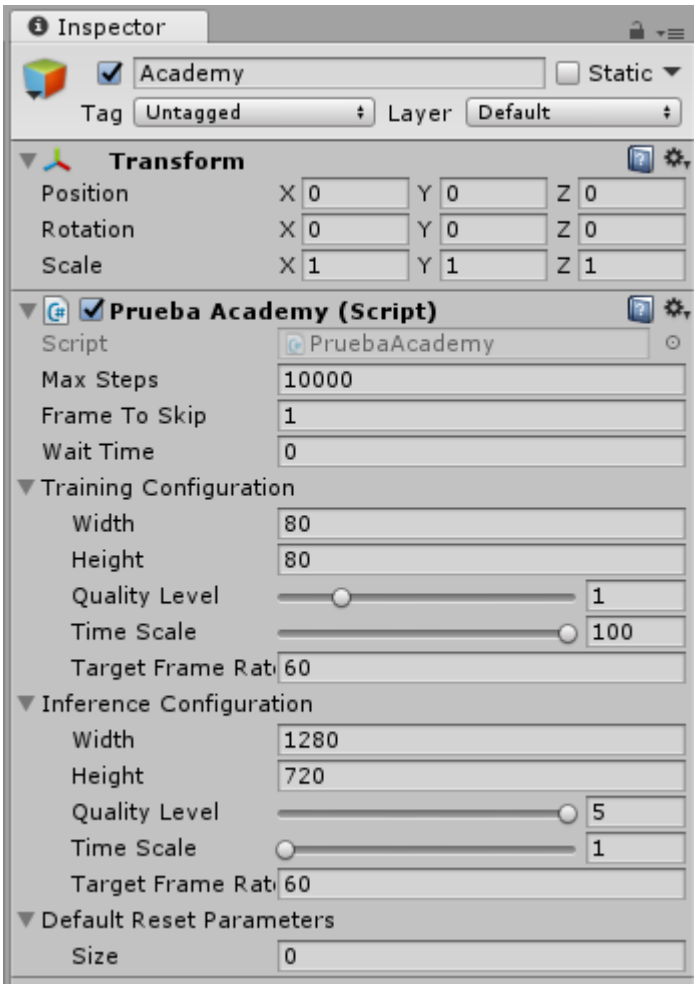


Ilustración 36. Inspector del objeto academia

8.1.10. Características de un cerebro

El cerebro encapsula el proceso de toma de decisiones. A cada agente se le debe asignar un cerebro, pero se puede usar el mismo cerebro con más de un agente.

En este caso se puede utilizar la clase *Brain* directamente, en lugar de una subclase. El comportamiento del cerebro está determinado por el tipo de cerebro. Durante el entrenamiento, se debe establecer el tipo de cerebro a *External* (externo). Una vez generado el modelo entrenado, para poder usarlo, se tiene que importar el archivo del modelo al proyecto de Unity y cambiar el tipo de cerebro a *Internal* (interno).

La clase *Brain* tiene varias propiedades importantes que se pueden establecer mediante la ventana del inspector. Estas propiedades deben ser apropiadas para los

agentes que utilizan el cerebro. Por ejemplo, la propiedad del tamaño del vector de observaciones debe coincidir exactamente con el tamaño del vector de un agente.

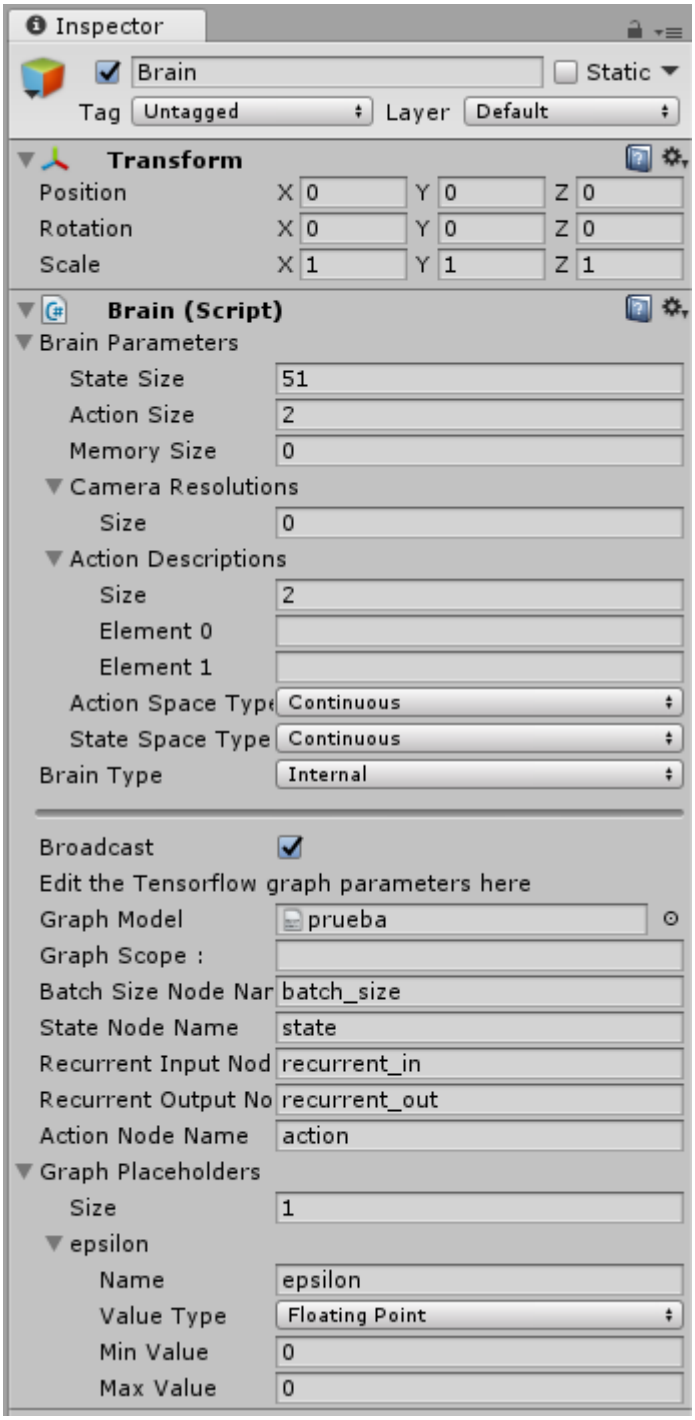


Ilustración 37. Inspector de un objeto cerebro

8.1.11. Características de un agente

La clase *Agent* representa un agente en la escena que recopila observaciones y lleva a cabo acciones.

Para crear un agente, se debe crear una subclase de *Agent* e implementar los métodos *CollectObservations()* y *AgentAction()*:

- *CollectObservations()*: recopila la observación del agente de su entorno.
- *AgentAction()*: realiza la acción elegida por el cerebro del agente y asigna una recompensa al estado actual.

Esta subclase se adjunta al *GameObject* de la escena que representa al agente (por ejemplo, un automóvil en una simulación de conducción).

También se debe determinar cómo un agente termina su tarea o agota el tiempo. Se puede establecer un agente a estado finalizado, en su función *AgentAction()*, cuando este haya terminado (o fallado) su tarea.

Se puede establecer la propiedad *Max Steps* (número máximo de pasos) del agente a un valor positivo y el agente se considerará finalizado después de haber realizado el número de pasos establecido. El próximo episodio comienza cuando la academia alcanza su propio número de *Max Steps*. Si se establece la propiedad *ResetOnDone* de un agente a verdadero, entonces el agente puede intentar su tarea varias veces en un episodio. También se puede utilizar la función *Agent.AgentReset()* para especificar el comportamiento del agente cuando se reinicie.

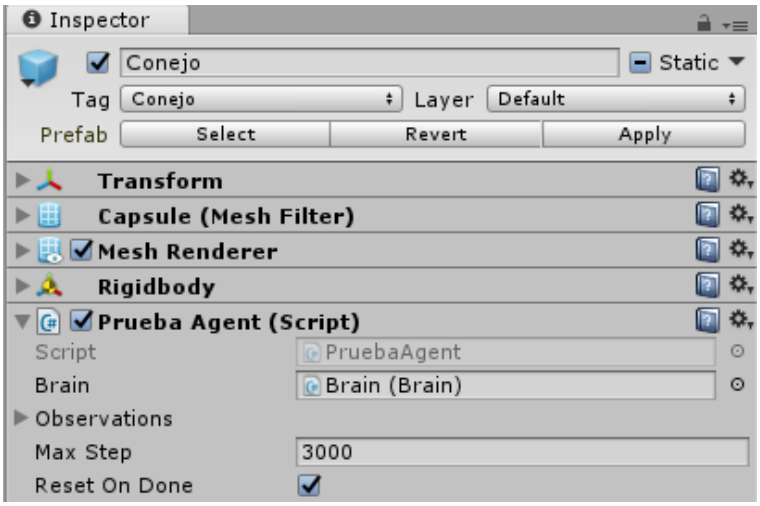


Ilustración 38. Inspector de un objeto agente

8.2. Entrenamiento externo

Una vez que se ha implementado el entorno de aprendizaje, se puede llevar a cabo el entrenamiento externo. Para ejecutar el entrenamiento de un agente, lo primero que hay que hacer es establecer el tipo de cerebro a *External*. Se tiene que compilar la escena de entrenamiento y guardar el fichero resultante en el directorio /python del proyecto.

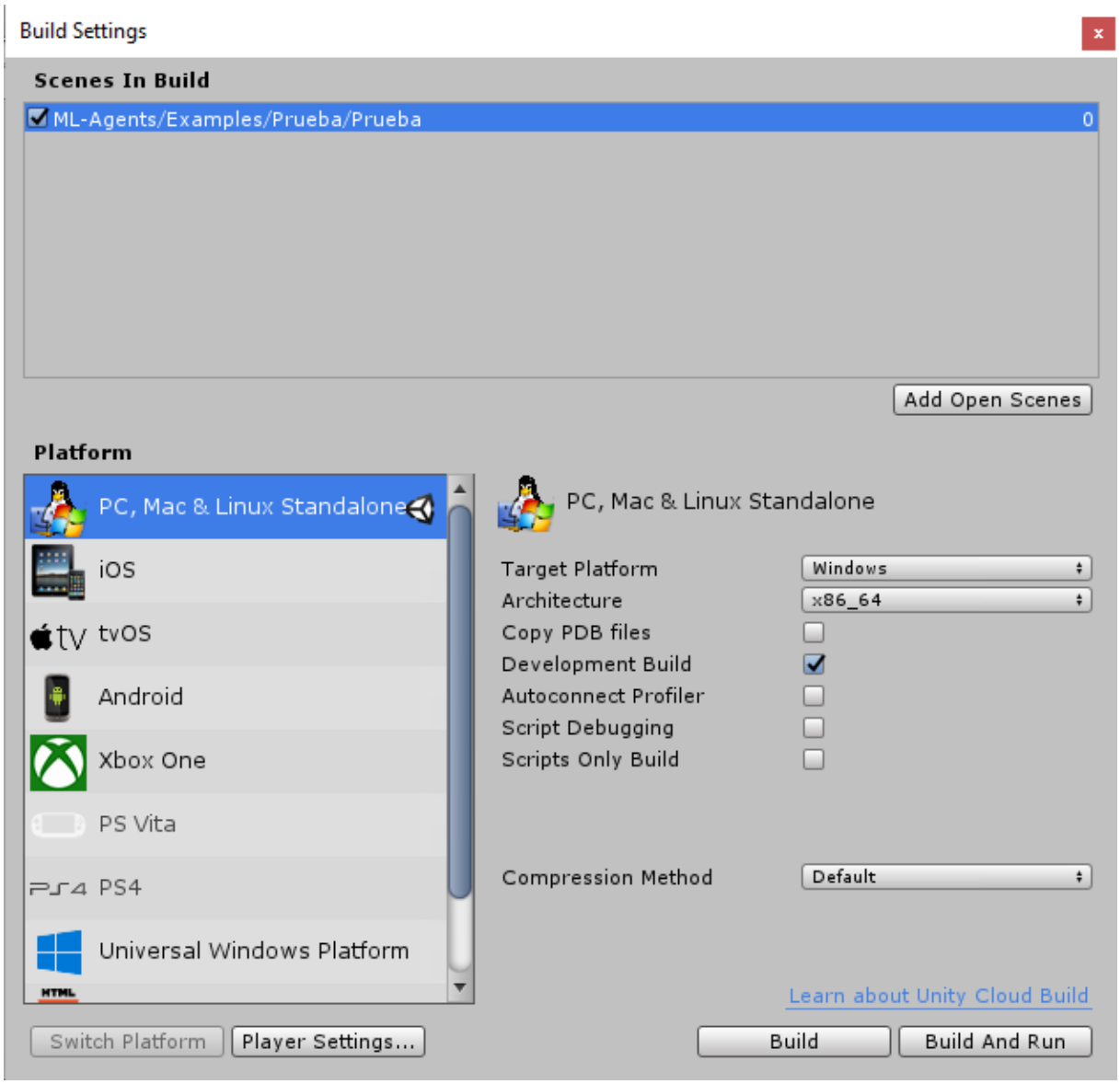


Ilustración 39. Ajustes de compilación

Tras esto, con Anaconda Prompt, hay que situarse en el directorio de python dentro de la carpeta *ml-agents* y ejecutar Jupyter Notebook.

```
(C:\ProgramData\Anaconda3) C:\Users\jose_\OneDrive\Documentos\Unity\Proyecto\python>jupyter notebook
```

Ilustración 40. Anaconda Prompt

Se abrirá una página mostrando el directorio python, donde se tendrá que seleccionar el algoritmo a utilizar. En este caso PPO.ipynb, que contiene el algoritmo Proximal Policy Optimization (PPO).

Se mostrará el documento Notebook del algoritmo, donde se pueden establecer varios hiperparámetros antes de la ejecución. Lo principal es escribir el nombre de la escena de entrenamiento previamente guardada.

Hyperparameters

```
In [8]: ### General parameters
max_steps = 5e5 # Set maximum number of steps to run environment.
run_path = "ppo" # The sub-directory name for model and summary statistics
load_model = False # Whether to load a saved model.
train_model = True # Whether to train the model.
summary_freq = 10000 # Frequency at which to save training statistics.
save_freq = 50000 # Frequency at which to save model.
env_name = "prueba" # Name of the training environment file.
curriculum_file = None
```

Ilustración 41. Parámetros generales

Otro hiperparámetro importante es *max_steps*, que indica el número de pasos del entrenamiento. Cuanto más alto sea este número mayor será el tiempo de entrenamiento.

Hyperparameters

```
In [6]: ### General parameters
max_steps = 5e5 # Set maximum number of steps to run environment.
run_path = "ppo" # The sub-directory name for model and summary statistics
load_model = False # Whether to load a saved model.
train_model = True # Whether to train the model.
summary_freq = 10000 # Frequency at which to save training statistics.
save_freq = 50000 # Frequency at which to save model.
env_name = "prueba" # Name of the training environment file.
curriculum_file = None

### Algorithm-specific parameters for tuning
gamma = 0.99 # Reward discount rate.
lambd = 0.95 # Lambda parameter for GAE.
time_horizon = 2048 # How many steps to collect per agent before adding to buffer.
beta = 1e-3 # Strength of entropy regularization
num_epoch = 5 # Number of gradient descent steps per batch of experiences.
num_layers = 2 # Number of hidden layers between state/observation encoding and value/policy layers.
epsilon = 0.2 # Acceptable threshold around ratio of old and new policy probabilities.
buffer_size = 2048 # How large the experience buffer should be before gradient descent.
learning_rate = 3e-4 # Model learning rate.
hidden_units = 64 # Number of units in hidden layer.
batch_size = 64 # How many experiences per gradient descent update step.
normalize = False
```

Ilustración 42 Hiperparámetros por defecto

Tras establecer los hiperparámetros hay que ejecutar hasta el apartado *Train the agents*.

Una vez que haya terminado el entrenamiento se exportará el **modelo entrenado** automáticamente. O bien se puede parar la ejecución antes de tiempo y ejecutar la parte llamada *Export.*

El resultado se exporta en *python/models/ppo* como un fichero de tipo .bytes. Es necesario copiarlo a la carpeta *TFModels* del proyecto, cambiar el cerebro a tipo *Internal* y elegirlo como *Graph model*.

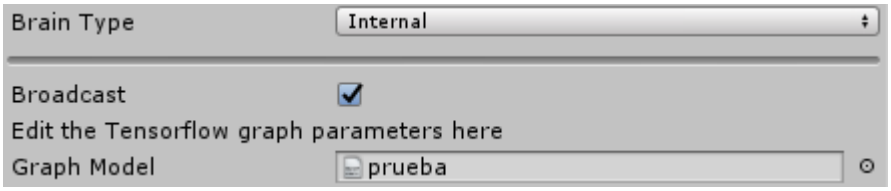


Ilustración 43. Ajustes del cerebro

Finalmente, ejecutando la escena de Unity se puede observar el comportamiento obtenido.

8.3. Implementación del entorno de entrenamiento

Para recrear mBiti World mediante ML-Agents, el primer paso consiste en entrenar un agente en un entorno de entrenamiento para así obtener un comportamiento inteligente.

ML-Agents aconseja empezar por lo básico. En este caso sería conseguir que un agente se dirija a un objetivo definido desde el principio. Esto representaría un animal que se dirige hacia la comida.

Para la fase de entrenamiento he tenido que implementar la mecánica del entorno de entrenamiento, las observaciones, acciones y recompensas.

El entorno de entrenamiento inicial consistía en lo siguiente:

- **Objetivo**: el agente debe desplazarse hacia la posición de la comida.
- **Área** de entrenamiento: dos *GameObjects* representan el agente y el objetivo. Sus posiciones iniciales son aleatorias.
- **Agentes**: un solo agente unido a un cerebro.
- **Función recompensa**:
    - -0.01 en cada paso. ML-Agents recomienda utilizar una recompensa negativa pequeña en cada paso en escenarios en los

que el comportamiento óptimo del agente es completar un episodio lo más rápido posible [29].
    - +1 al alcanzar el objetivo.

- **Cerebro**: un cerebro con los siguientes espacios de observaciones y acciones:
    - Espacio de **observaciones** (Continuo): vector de tamaño 4 correspondiente a:
        - Dirección hacia el objetivo.
        - Dirección hacia la que se dirige el agente.
    - Espacio de **acciones** (Discreto): tamaño 5. Correspondiente al movimiento en el plano XZ y no moverse:
        - -1 en X
        - +1 en X
        - -1 en Z
        - +1 en Z
        - No hacer nada
- **Reinicio**: comienza un nuevo episodio del entrenamiento cuando el agente alcanza el objetivo o cuando se supera el número de pasos por episodio establecidos.

Este se trata de un entorno totalmente observable ya que se conocen todos los aspectos que son relevantes en la toma de decisiones.

Una vez implementada la mecánica del entrenamiento se puede ejecutar el entrenamiento externo, tal y como he detallado previamente. Tras esto se puede ver el comportamiento al ejecutar la escena de Unity.

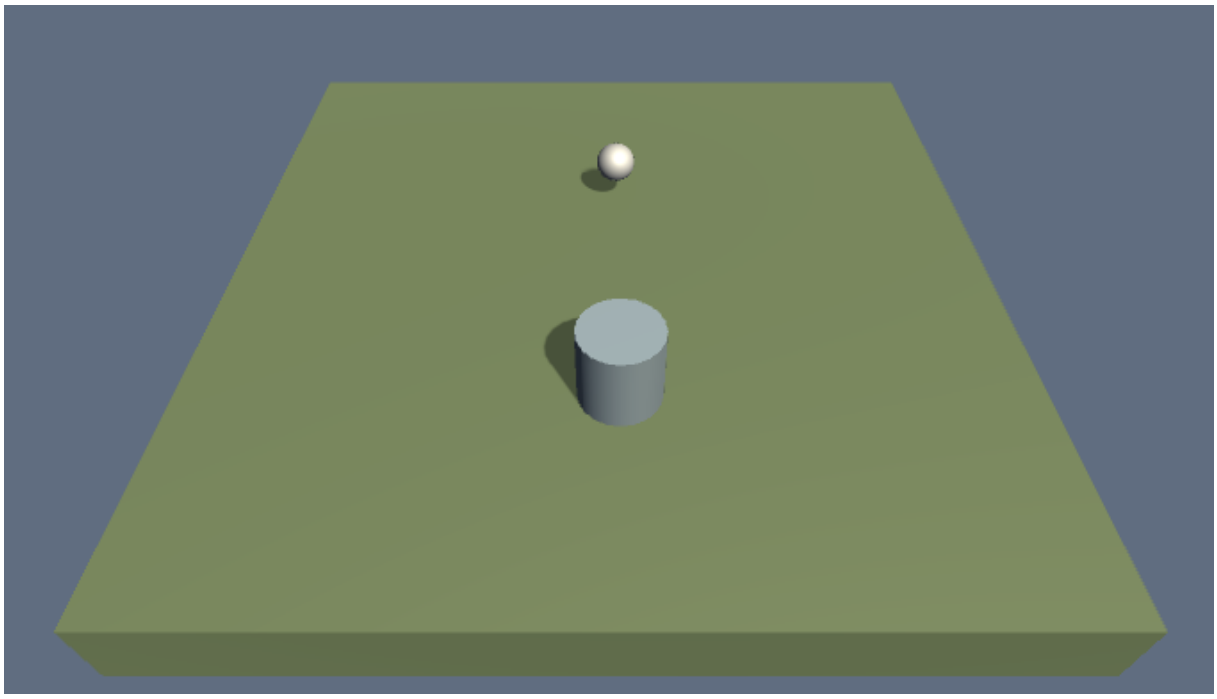


Ilustración 44. Entrenamiento básico. Agente (cilindro) se dirige a objetivo (esfera).

El comportamiento que obtuve en todas las ejecuciones era el esperado. Conociendo la posición de la comida, el agente se dirige con éxito hacia esta.

Sin embargo, esto es solo el comportamiento básico. Según los requisitos para la aplicación, los animales no pueden saber la posición de la comida hasta que esta haya entrado en su rango de visión. Por lo tanto, el entorno será parcialmente observable.

Por otro lado, el movimiento del agente con un espacio de acciones discreto no permite al agente moverse de manera fluida, por lo que para un animal es más adecuado un espacio de direcciones continuo.

He modificado lo siguiente:

- **Área** de entrenamiento: He incluido paredes para evitar que el agente se salga del área.
- **Cerebro**: un cerebro con los siguientes espacios de observaciones y acciones:
    - Espacio de **observaciones** (Continuo):
        - Dirección hacia la que se dirige el agente.

- Booleano que indica si el agente ha visto la comida. La variable será verdadera si la comida colisiona con un *Collider Trigger* que representa el rango de visión.
- Dirección hacia el objetivo. Se pasará si el booleano anterior es true.

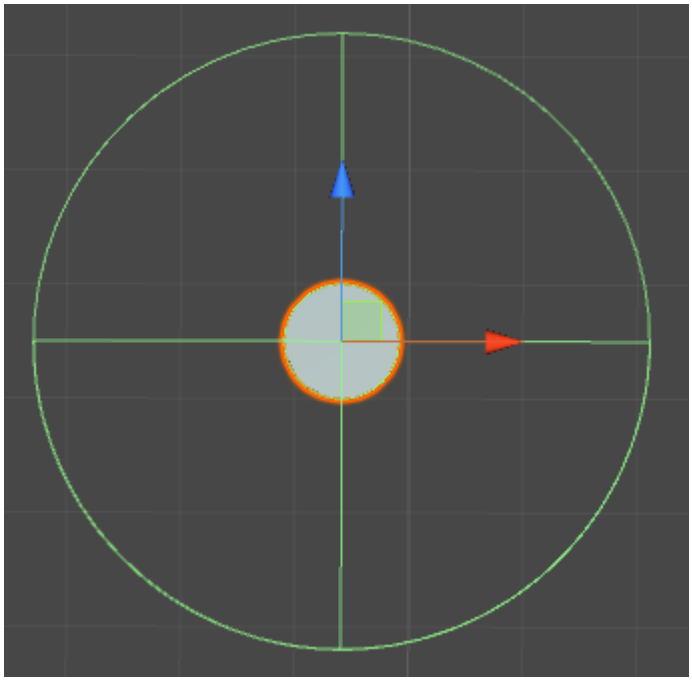


Ilustración 45. Rango de visión

- Espacio de **acciones** (Continuo): tamaño 2. Correspondiente al movimiento en una dirección concreta o no moverse si es cero:
  - De 1 a -1 en X
  - De 1 a -1 en Z

El comportamiento obtenido tras el entrenamiento no era el deseado. Esta vez el agente solo se mueve en una dirección concreta y acaba parado contra la pared.

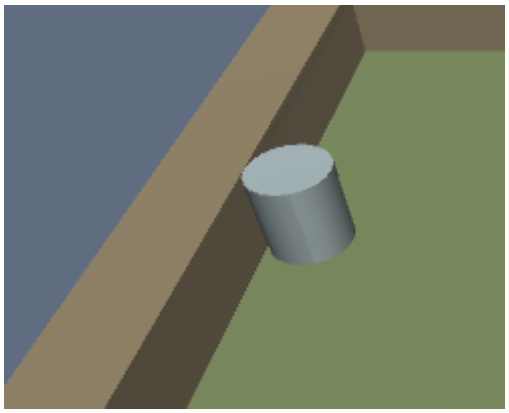


Ilustración 46. Comportamiento no deseado

8.3.1. Problemas en el entrenamiento

Para intentar mejorar el comportamiento obtenido hice pruebas con la función recompensa. Añadí una pequeña recompensa positiva cuando el agente se moviera para evitar que se quedase parado. Pero esto solo consiguió que el agente se quedara en la misma zona moviéndose hacia delante y hacia atrás, al proporcionarle esto recompensa positiva y “olvidarse” de la recompensa por alcanzar la comida. Por lo tanto, esta recompensa la eliminé.

Lo siguiente que probé fue añadir una pequeña recompensa negativa por colisionar con cualquier obstáculo que no fuera la comida. El comportamiento obtenido no mejoró apenas, el agente cambiaba de dirección, pero solía quedarse en la misma zona o quedarse estancado en las esquinas.

Tras esto decidí añadir una recompensa negativa para cuando se quede parado y una positiva si el agente no se encuentra cerca de donde estaba unos segundos antes. Esto no consiguió tampoco mejorar el comportamiento.

Posteriormente decidí modificar el área de entrenamiento, haciéndola más grande, incluyendo obstáculos y más comida. Pero el comportamiento seguía igual.

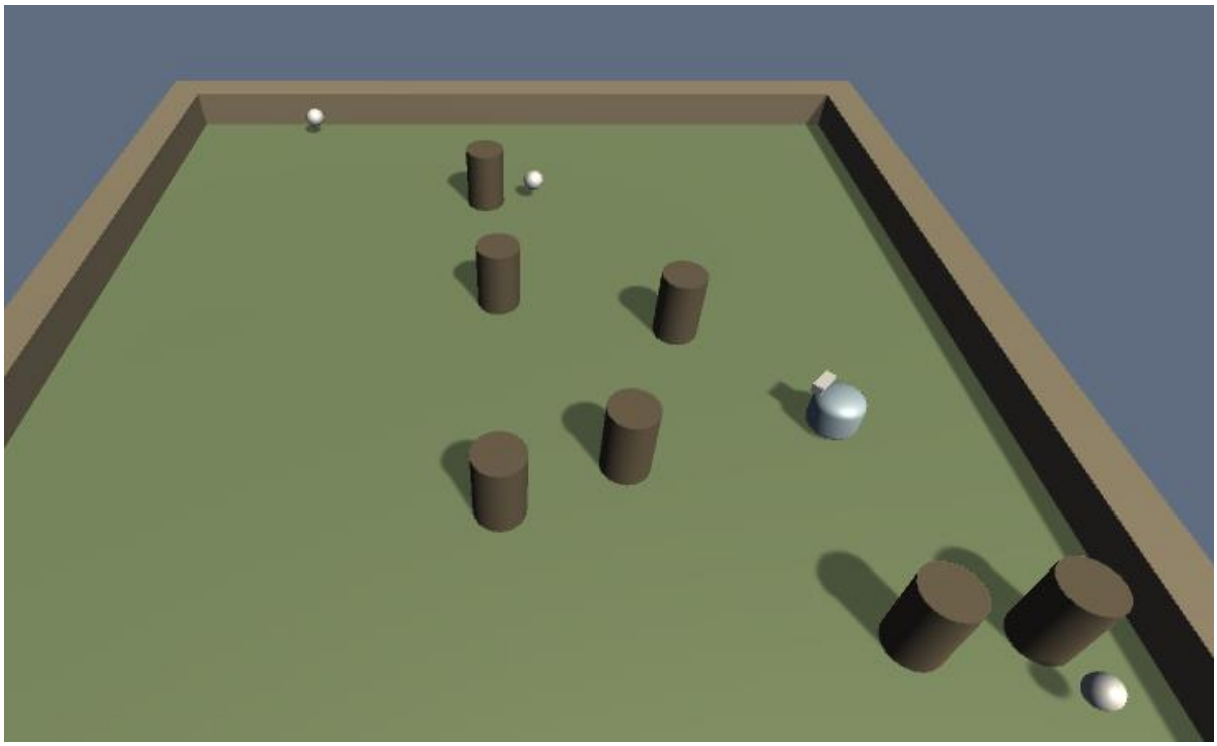


Ilustración 47. Entorno de entrenamiento

Al mismo tiempo también estuve ajustando los hiperparámetros del algoritmo, como el número de pasos del entrenamiento, para que fueran los adecuados para una tarea compleja. Por defecto el número de pasos es 500000, lo que lleva una ejecución del entrenamiento de una media hora. Lo aumenté a 10000000 y comprobé que se obtenía una mejora en la recompensa acumulada alrededor del paso 8000000.

```
### General parameters
max_steps = 1e7 # Set maximum number of steps to run environment.
run_path = "ppo" # The sub-directory name for model and summary statistics
load_model = False # Whether to load a saved model.
train_model = True # Whether to train the model.
summary_freq = 10000 # Frequency at which to save training statistics.
save_freq = 50000 # Frequency at which to save model.
env_name = "prueba" # Name of the training environment file.
curriculum_file = None
```

Ilustración 48. Hiperparámetros - Parámetros generales

Al aumentar el tiempo de entrenamiento se obtienen mejores resultados. El inconveniente de esto es que un entrenamiento de diez millones de pasos tarda alrededor de diez horas.

El comportamiento obtenido era más cercano al deseado, el agente exploraba más cantidad del área, pero solía chocarse varias veces con el mismo obstáculo y seguía estancándose en las esquinas.

Lo siguiente que decidí fue aumentar la información que el agente obtiene del entorno. Decidí utilizar *Ray casting* para la visión del agente, utilizando varios rayos con origen en el agente y distancia determinada. Comprobaría si colisionan con algún objeto e informaría al agente de la distancia a la que se encuentra este objeto. También obtendría recompensa negativa al reducirse demasiado la distancia a un obstáculo.

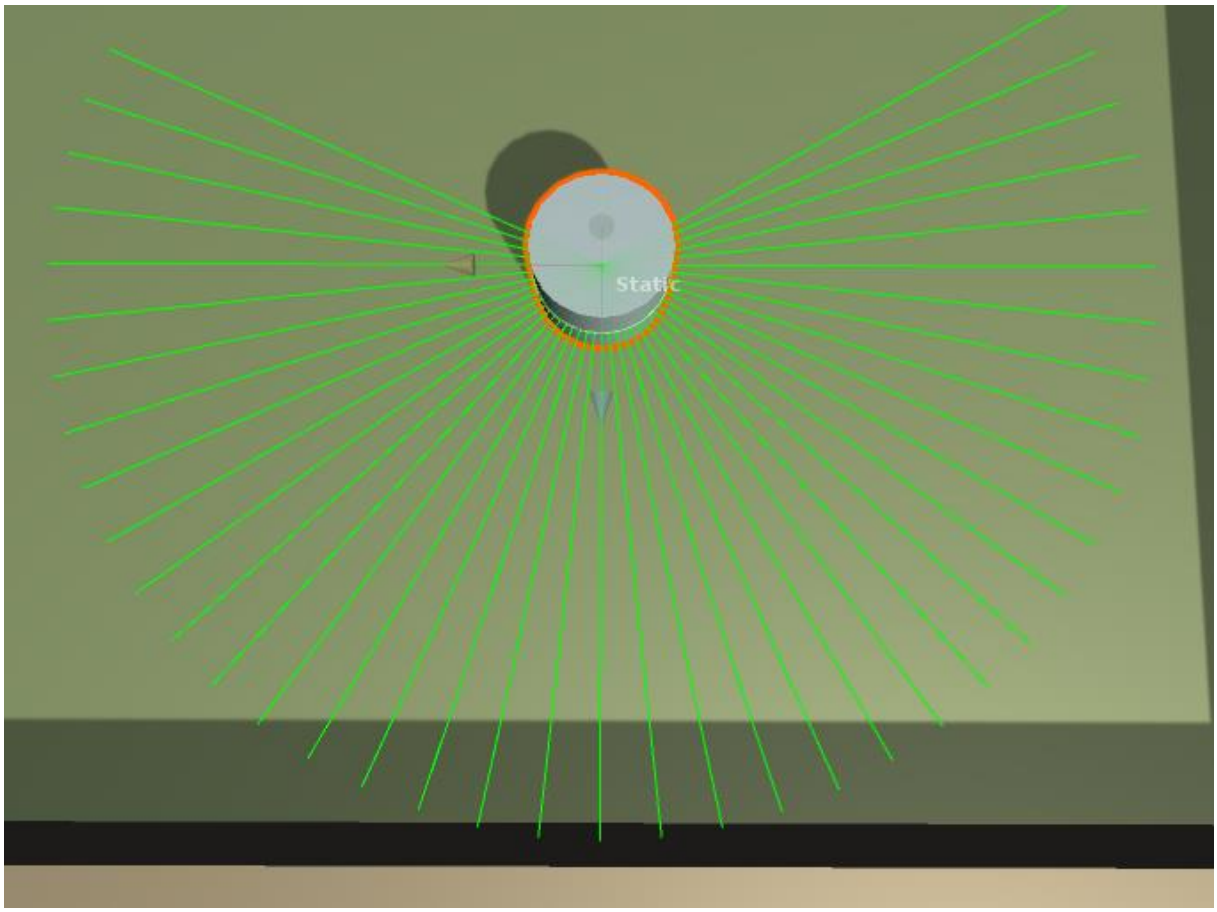


Ilustración 49. Visión mediante Raycasting

El comportamiento obtenido mejoró, el agente ya no solía colisionar con los obstáculos o la pared al acercarse a ellos, pero seguía quedándose estancado en las esquinas.

Lo que consiguió evitar esto fue aumentar la recompensa negativa obtenida al quedarse parado o al acercarse demasiado a un obstáculo.

8.3.2. Entorno de entrenamiento final

Tras resolver los principales problemas, implementé la funcionalidad de depredador, para poder realizar un entrenamiento paralelo con varios agentes, unos que representasen conejos y otros, lobos. En este caso, el objetivo de un depredador es alcanzar al conejo, mientras que el conejo, además de obtener su comida (una planta), debe aprender a evitar al lobo. Para conseguir esto le proporciono recompensa negativa al conejo si el lobo se encuentra en su rango de visión.

A continuación detallo el entorno de entrenamiento final:

- **Objetivo**: cada agente debe desplazarse hacia la posición de su comida.
- **Área** de entrenamiento: *GameObjects* que representan a conejos, lobos y plantas (comida para el conejo). Sus posiciones iniciales son aleatorias. El área tiene paredes alrededor y obstáculos cuyas posiciones son también aleatorias.
- **Agentes**: dos conejos y dos lobos unidos al mismo cerebro.
- **Función recompensa**:
    - -0.005 en cada paso. ML-Agents recomienda utilizar una recompensa negativa pequeña en cada paso en escenarios en los que el comportamiento óptimo del agente es completar un episodio lo más rápido posible.
    - +1 al alcanzar el objetivo. Según ML-Agents, la magnitud de cualquier recompensa dada no debe ser mayor a 1 para asegurar un proceso de aprendizaje más estable.
    - -0.05 si se queda parado.
    - +0.005 si no está cerca de la posición en la que estaba unos segundos antes.
    - -0.005 si una vez visto el objetivo, se dirige en dirección contraria.
    - -0.01 por cada rayo cuya distancia de colisión es inferior a un límite.
    - +0.0001 por cada rayo que no colisiona.

  - -0.005 si el agente es un conejo y el lobo está dentro de su rango de visión.

- **Cerebro**: un cerebro con los siguientes espacios de observaciones y acciones:
  - Espacio de **observaciones** (Continuo): vector de tamaño 51 correspondiente a lo siguiente:
    - Booleano que indica si el agente ha visto la comida.
    - Dirección hacia el objetivo. Se pasará si el booleano anterior es true.
    - Distancia entre el agente el objetivo. Se pasará si el primer booleano es true.
    - Dirección hacia la que se dirige el agente.
    - Velocidad del agente.
    - Vector de tamaño 40 con información sobre cada rayo: 1 si el rayo no colisiona o la distancia (normalizada) al objeto con el que colisiona.
    - Distancia entre conejo y lobo (si el agente es un conejo).
    - Orientación del agente.
  - Espacio de **acciones** (Continuo): tamaño 2. Correspondiente al movimiento en una dirección concreta (o no moverse si es un vector nulo):
    - De 1 a -1 en X
    - De 1 a -1 en Z
- **Reinicio**: comienza un nuevo episodio del entrenamiento cuando un agente alcanza su objetivo o cuando se supera el número de pasos por episodio establecidos.

8.3.3. Ajuste de hiperparámetros

Además del número de pasos, para conseguir buenos resultados, también es necesario ajustar otros hiperparámetros de entrenamiento [21].

Los hiperparámetros de entrenamiento son los siguientes:

- **Gamma**. Se define como el factor de descuento para recompensas futuras. Esto se refiere al periodo de tiempo en el que el agente no debería preocuparse por las posibles recompensas. En situaciones en las que el agente debería recibir recompensas en un futuro lejano, este valor debería ser grande. En los casos en que las recompensas son más inmediatas, puede ser más pequeño. El rango típico es de 0,8 a 0,995.
- **Lambda**. Se puede considerar como cuánto depende el agente de su estimación de valor actual al calcular una estimación de valor actualizada. Los valores bajos corresponden a confiar más en la estimación del valor actual y los valores altos corresponden a confiar más en las recompensas recibidas en el entorno. El rango típico es de 0,9 a 0,95.
- **Buffer size**. Corresponde a la cantidad de experiencias (observaciones del agente, acciones y recompensas obtenidas) que deben recopilarse antes de realizar cualquier aprendizaje o actualización del modelo. Este debe ser un múltiplo del *batch size*. Normalmente, un tamaño más grande corresponde a actualizaciones de entrenamiento más estables. El rango típico es de 2048 a 409600.
- **Batch size**. Es el número de experiencias utilizadas para una iteración de actualización de descenso de gradiente. Debe ser una fracción del *buffer size*. Si se está utilizando espacio de acciones continuo, este valor debe ser mayor que si se está utilizando un espacio de acciones discreto. El rango típico es de 512 a 5120 para un espacio de acciones continuo y de 32 a 512 para un espacio de acciones discreto.
- **Number of Epochs**. Es el número de pasadas a través del búfer de experiencia durante el descenso de gradiente. Mientras más grande sea *batch size*, mayor podrá ser este número. Disminuir el número asegurará actualizaciones más estables, a costa de un aprendizaje más lento. El rango típico es de 3 a 10.
- **Learning Rate**. Corresponde a la tasa de aprendizaje del descenso de gradiente. Debería reducirse si el entrenamiento es inestable y la recompensa no aumenta constantemente. El rango típico es de 1e-5 a 1e-3.

- **Time horizon**. Se define como la cantidad de pasos de experiencia a recopilar por agente antes de agregarlo al búfer de experiencia. Cuando se alcanza este límite antes del final de un episodio, se utiliza una estimación de valor para predecir la recompensa esperada del estado actual del agente. En los casos donde hay recompensas frecuentes dentro de un episodio, o los episodios son extremadamente grandes, un número más pequeño puede ser más apropiado. El rango típico es de 32 a 2048.
- **Max Steps**. Corresponde a cuántos **pasos** de la simulación (multiplicados por *frame-skip*) se ejecutan durante el proceso de entrenamiento. Este valor debe aumentarse para problemas más complejos. El rango típico es de 5e5 a 1e7.
- **Beta**. Se define como la fuerza de regularización de la entropía, es decir, lo que hace que la política sea "más aleatoria". Esto garantiza que los agentes exploren adecuadamente el espacio de acciones durante el entrenamiento. Aumentar este parámetro asegurará que se tomen más acciones al azar. Se debe ajustar de manera que la **entropía** (variable medible desde TensorBoard) disminuya lentamente junto con los aumentos en la recompensa. Si la entropía decrementa demasiado rápido, se debería aumentar *beta*. Si lo hace muy lentamente, se debería disminuir. El rango típico es de 1e-4 a 1e-2.
- **Epsilon**. Corresponde al umbral aceptable de divergencia entre las políticas antiguas y las nuevas durante la actualización del descenso de gradiente. Un valor bajo dará como resultado actualizaciones más estables, pero también ralentizará el proceso de entrenamiento. El rango típico es de 0,1 a 0,3.
- **Normalize**. Indica si la normalización se aplica a las entradas de observaciones. La normalización puede ser útil en casos con problemas complejos de control continuo, pero puede ser perjudicial con problemas más simples de control discreto. Normalize es un booleano.
- **Number of Layers**. Se define como la cantidad de **capas ocultas** (de la red neuronal) presentes después de la entrada de observaciones. Para problemas simples, es probable que con menos capas se entrene más

rápido y eficientemente. Para problemas de control más complejos, se pueden necesitar más capas. El rango típico es de 1 a 3.

- **Hidden Units**. Corresponde a cuántas **unidades** hay en cada capa oculta de la red neuronal. Para problemas simples donde la acción correcta es una combinación directa de las entradas de observaciones, este valor debería ser pequeño. Para problemas donde la acción es una interacción compleja entre las variables de observación, debería ser más grande. El rango típico es de 32 a 512.

A continuación se pueden observar los valores de hiperparámetros con los que se han obtenido los mejores resultados, después de realizar pruebas con diferentes configuraciones:

Hiperparámetro	Valor	Observaciones
Gamma	0.99	Este valor es grande porque el agente no obtiene recompensas inmediatas, al tener que explorar el entorno y buscar la comida
Lambda	0.95	Es un valor alto porque el agente debe ser más dependiente de las recompensas recibidas en el entorno
Buffer Size	10000	Un múltiplo de Batch size
Batch Size	2000	Al utilizar un espacio de acciones continuo, el valor recomendado está entre 512 y 5120
Number of Epochs	5	Un valor bajo, que asegura actualizaciones más estables, a costa de un aprendizaje más lento

Learning Rate	1e-5	Un número bajo asegura actualizaciones más estables
Time Horizon	2048	Un número adecuado para capturar todo el comportamiento importante dentro de una secuencia de acciones de un agente
Max Steps	1e7	Al ser un problema complejo, este debe ser un valor grande
Beta	1e-3	Valor ajustado de forma que la entropía disminuya lentamente junto con los aumentos en la recompensa
Epsilon	0.2	Un valor medio para la conseguir actualizaciones estables, pero no ralentizar demasiado el proceso de entrenamiento
Normalize	False	No es necesario la normalización de las observaciones
Number of Layers	3	Al ser un problema complejo se necesitan más capas (siendo el valor recomendado de 1 a 3)
Hidden Units	512	Al ser un problema donde la acción resulta de una interacción compleja entre

		las variables de observación, debe ser un valor grande.

Tabla 21. Hiperparámetros utilizados

Hyperparameters

```
### General parameters
max_steps = 1e7 # Set maximum number of steps to run environment.
run_path = "ppo" # The sub-directory name for model and summary statistics
load_model = False # Whether to load a saved model.
train_model = True # Whether to train the model.
summary_freq = 10000 # Frequency at which to save training statistics.
save_freq = 50000 # Frequency at which to save model.
env_name = "prueba" # Name of the training environment file.
curriculum_file = None

### Algorithm-specific parameters for tuning
gamma = 0.99 # Reward discount rate.
lambd = 0.95 # Lambda parameter for GAE.
time_horizon = 2048 # How many steps to collect per agent before adding to buffer.
beta = 1e-3 # Strength of entropy regularization
num_epoch = 5 # Number of gradient descent steps per batch of experiences.
num_layers = 3 # Number of hidden layers between state/observation encoding and value/policy layers.
epsilon = 0.2 # Acceptable threshold around ratio of old and new policy probabilities.
buffer_size = 10000 # How large the experience buffer should be before gradient descent.
learning_rate = 1e-5 # Model learning rate.
hidden_units = 512 # Number of units in hidden layer.
batch_size = 2000 # How many experiences per gradient descent update step.
normalize = False
```

Ilustración 50. Hiperparámetros establecidos

8.3.4. Análisis del entrenamiento

Durante el entrenamiento se puede observar la recompensa media por episodio, así como la desviación típica.

```
Saved Model
Step: 3960000. Mean Reward: -6.3756594366659405. Std of Reward: 7.2806621335160475.
Step: 3970000. Mean Reward: -6.373471136250707. Std of Reward: 8.43090179963951.
Step: 3980000. Mean Reward: -7.496682527791928. Std of Reward: 9.520814384484202.
Step: 3990000. Mean Reward: -6.8336068053771815. Std of Reward: 8.918309284185232.
Step: 4000000. Mean Reward: -11.319275255405971. Std of Reward: 10.880157268115624.
Saved Model
Step: 4010000. Mean Reward: -7.669794647458624. Std of Reward: 10.091473735583692.
Step: 4020000. Mean Reward: -5.508525810249923. Std of Reward: 8.613821950262858.
Step: 4030000. Mean Reward: -9.166976622950777. Std of Reward: 11.10602902069163.
Step: 4040000. Mean Reward: -5.887973137474981. Std of Reward: 8.375732671920801.
Step: 4050000. Mean Reward: -6.222928562892077. Std of Reward: 10.248542888419912.
```

Ilustración 51. Recompensa durante el entrenamiento

Para observar el proceso de entrenamiento en más detalle he utilizado el *Scalar Dashboard* de TensorBoard. Este permite visualizar datos del entrenamiento que varían con el tiempo [30]:

- **Recompensa acumulada**: recompensa acumulada media por episodio. Debería aumentar durante una sesión de entrenamiento satisfactoria.

Ilustración 52. Ejemplo de recompensa acumulada

- **Entropía**: indica cómo de aleatorias son las decisiones del modelo. Debería disminuir lentamente durante un proceso de entrenamiento exitoso. Si disminuye demasiado rápido, el hiperparámetro *beta* debería aumentarse.

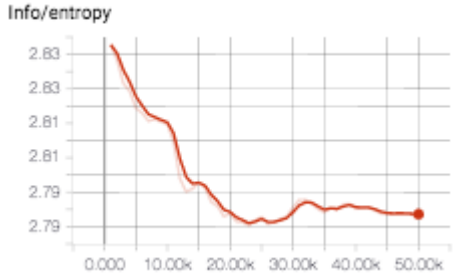


Ilustración 53. Ejemplo de entropía

- **Duración del episodio**: duración media de cada episodio en el entorno.

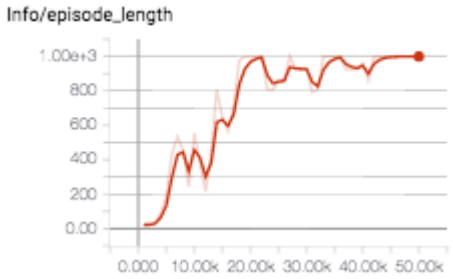


Ilustración 54. Ejemplo de duración del episodio

- **Tasa de aprendizaje**: tamaño de los pasos que toma el algoritmo de entrenamiento mientras busca la política óptima. Debería disminuir con el tiempo.

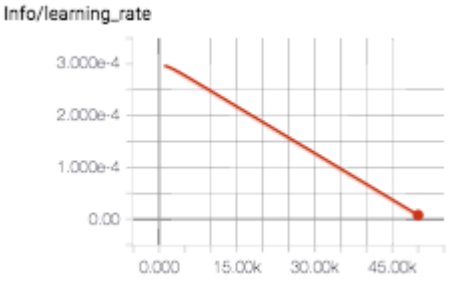


Ilustración 55. Ejemplo de tasa de aprendizaje

- **Pérdida de política**: pérdida media de la actualización de la función de política. Está relacionada con cuánto está cambiando la política. Su magnitud debería disminuir durante una sesión de entrenamiento satisfactoria.

Ilustración 56. Ejemplo de pérdida de política

- **Estimación del valor**: estimación del valor medio para todos los estados visitados por el agente. Debe aumentar durante una sesión de entrenamiento satisfactoria.

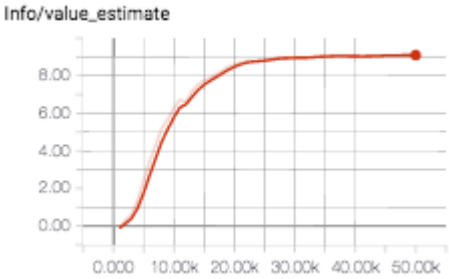


Ilustración 57. Ejemplo de estimación del valor

- **Pérdida del valor**: pérdida media de la actualización de la función de valor. Tiene relación con cómo de bien el modelo puede predecir el valor de cada estado. Debería disminuir durante una sesión de entrenamiento satisfactoria.

Ilustración 58. Ejemplo de pérdida del valor

A continuación detallo las estadísticas relativas al entrenamiento final:

Como se puede ver en la gráfica de la **recompensa acumulada**, esta va aumentando progresivamente hasta los 80000000 pasos (~8 horas). A partir de este momento la recompensa se mantiene prácticamente constante.

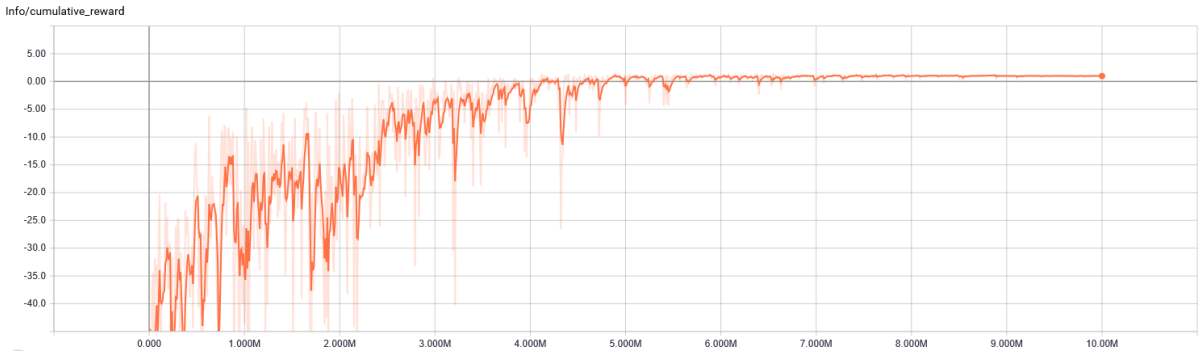


Ilustración 59. Recompensa acumulada por pasos

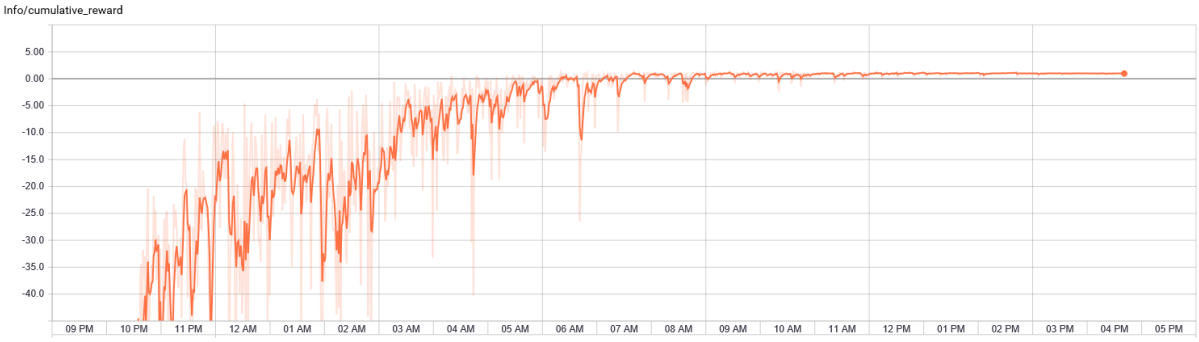


Ilustración 60. Recompensa acumulada por horas

La **entropía** disminuye lentamente durante el proceso de entrenamiento. Esto indica que el hiperparámetro *beta* establecido es adecuado.

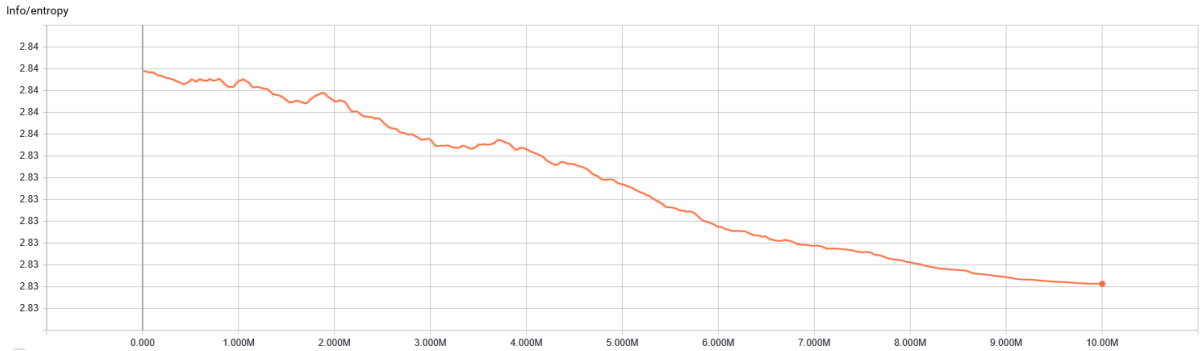


Ilustración 61. Entropía

La siguiente gráfica muestra la **duración** media de cada **episodio**.

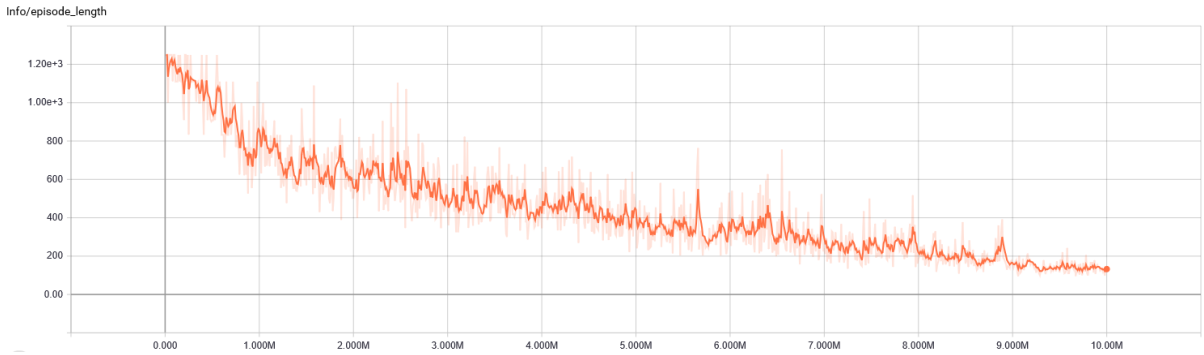


Ilustración 62. Duración del episodio

La **tasa de aprendizaje** disminuye con el tiempo, tal y como debe hacer en una sesión de entrenamiento satisfactoria.

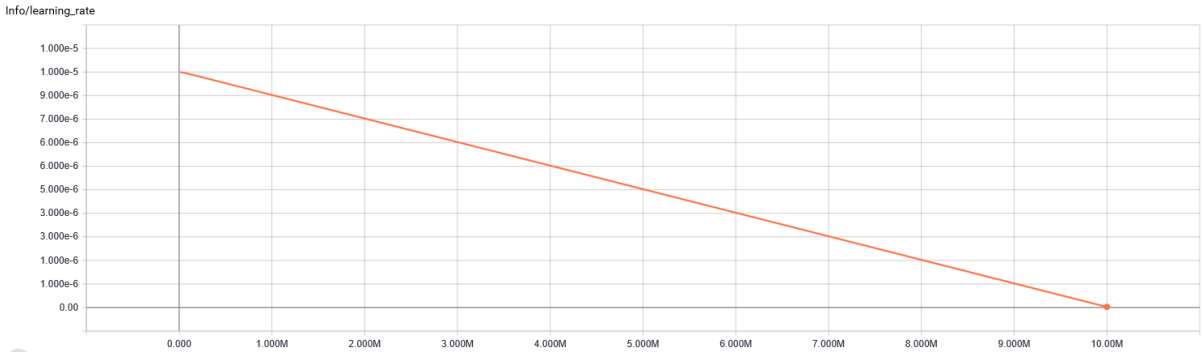


Ilustración 63. Tasa de aprendizaje

En la gráfica de la **pérdida de política** se puede observar como la política varía mucho a lo largo del entrenamiento mientras que la magnitud va disminuyéndose al final del entrenamiento.

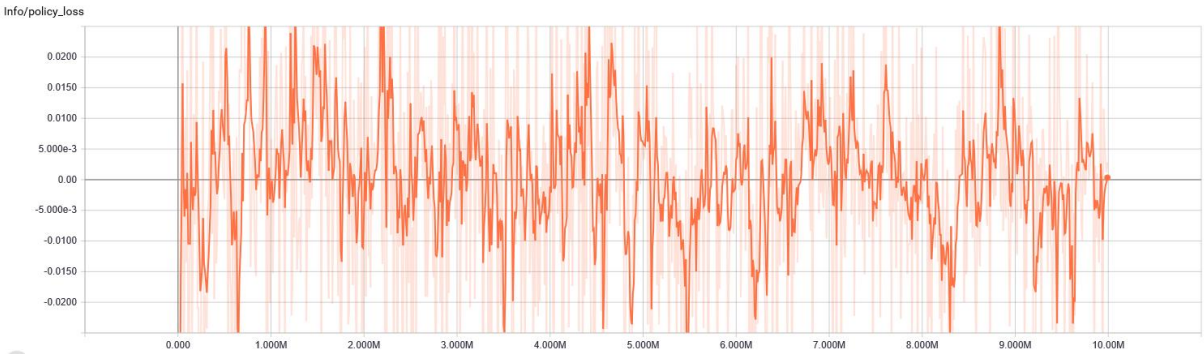


Ilustración 64. Pérdida de política

La **estimación del valor** aumenta durante la sesión de entrenamiento, que es lo esperado.

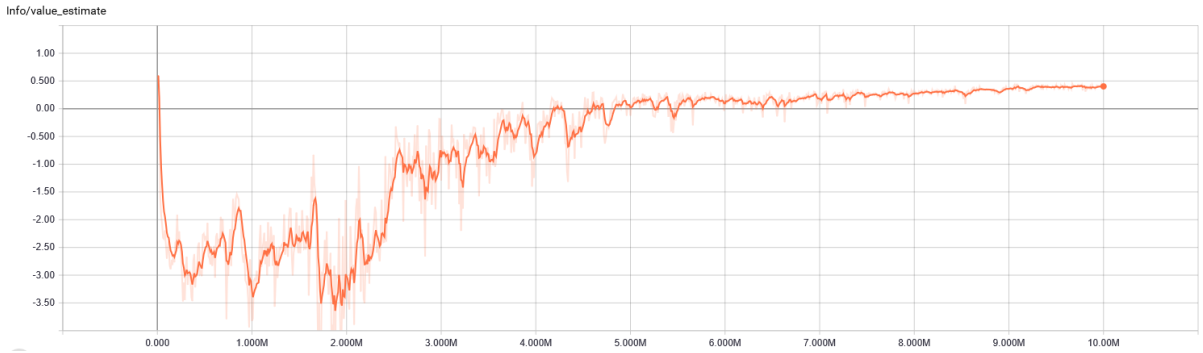


Ilustración 65. Estimación del valor

La **pérdida del valor** disminuye durante la sesión de entrenamiento. Esto es también lo que se espera de un buen entrenamiento.

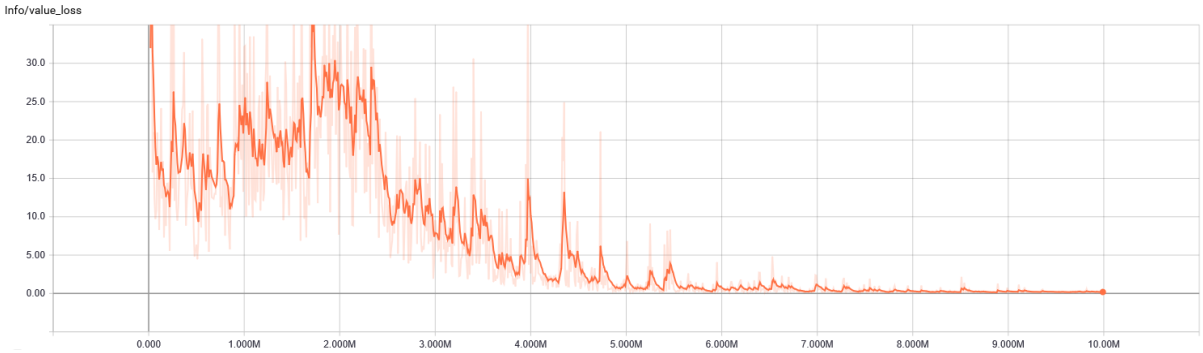


Ilustración 66. Pérdida del valor

8.4. Implementación del juego

Tras haber obtenido un modelo con un comportamiento inteligente mediante ML-Agents, puedo pasar a desarrollar una aplicación (juego) basada en mBiti World en la que utilizar este modelo.

8.4.1. Escena del juego

La escena en la que se desarrolla el juego simula un bosque con plantas y animales.

He utilizado la herramienta *Terrain* [31] de Unity para aplicar texturas al terreno, añadir hierba y cambiar la altura del terreno. Todo esto se realiza mediante un sistema de pinceles.

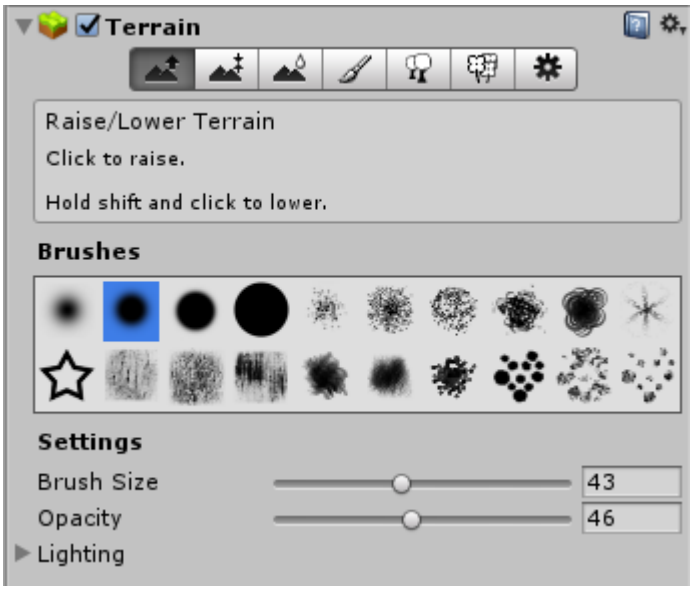


Ilustración 67. Terreno en el inspector

Los árboles (que representan los obstáculos del entorno) y las plantas (la comida de los herbívoros) se instancian aleatoriamente al inicio de cada partida. La hierba se crea a partir de *sprites*. Los árboles y plantas son modelos 3D.

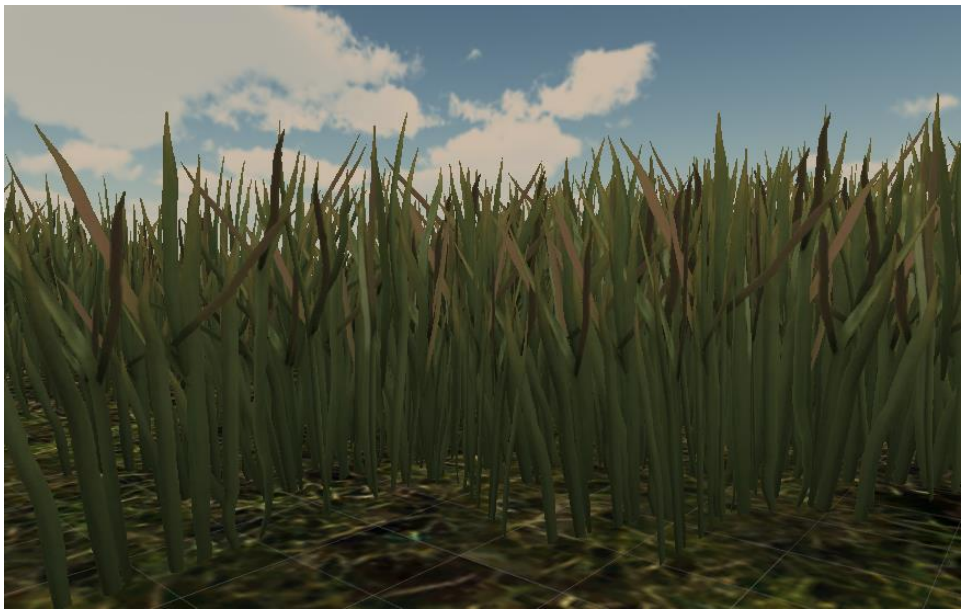


Ilustración 68. Terreno y Skybox

También he utilizado un *Skybox* [32], que simula un cielo aplicando texturas a una geometría envolvente.

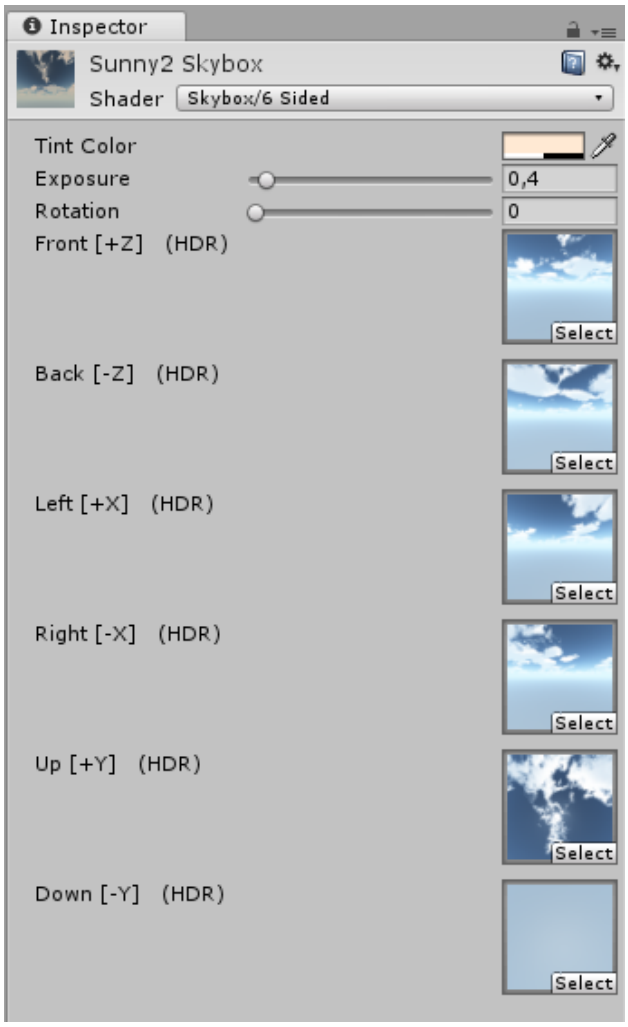


Ilustración 69. Skybox en el inspector

Los árboles, la hierba y las texturas las he descargado de la Asset Store, una biblioteca de recursos (assets) para Unity. Las plantas las he obtenido del repositorio de modelos 3D free3d.com.

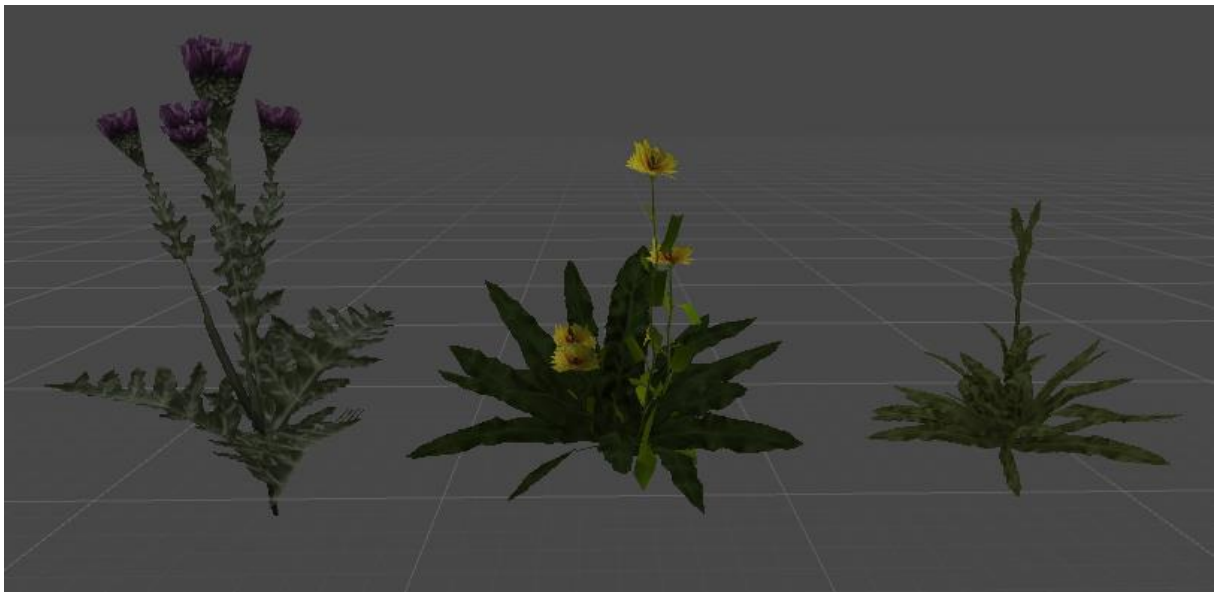


Ilustración 70. Plantas

8.4.2. Personajes y animaciones

Los personajes del juego son los animales. Estos consistirán en lobos como animales carnívoros y conejos como animales herbívoros.

Para esto he utilizado un *asset* de la Asset Store que contiene modelos 3D de estos animales, así como varias animaciones.

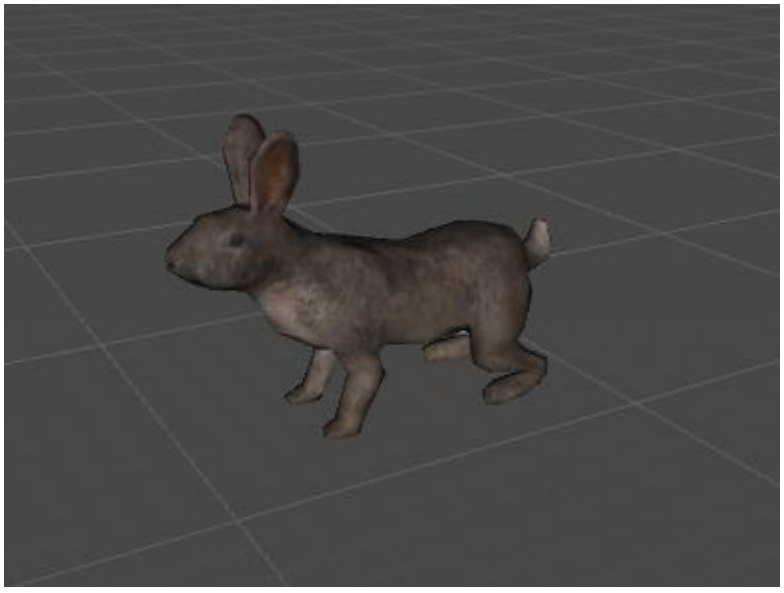


Ilustración 71. Modelo 3D de conejo

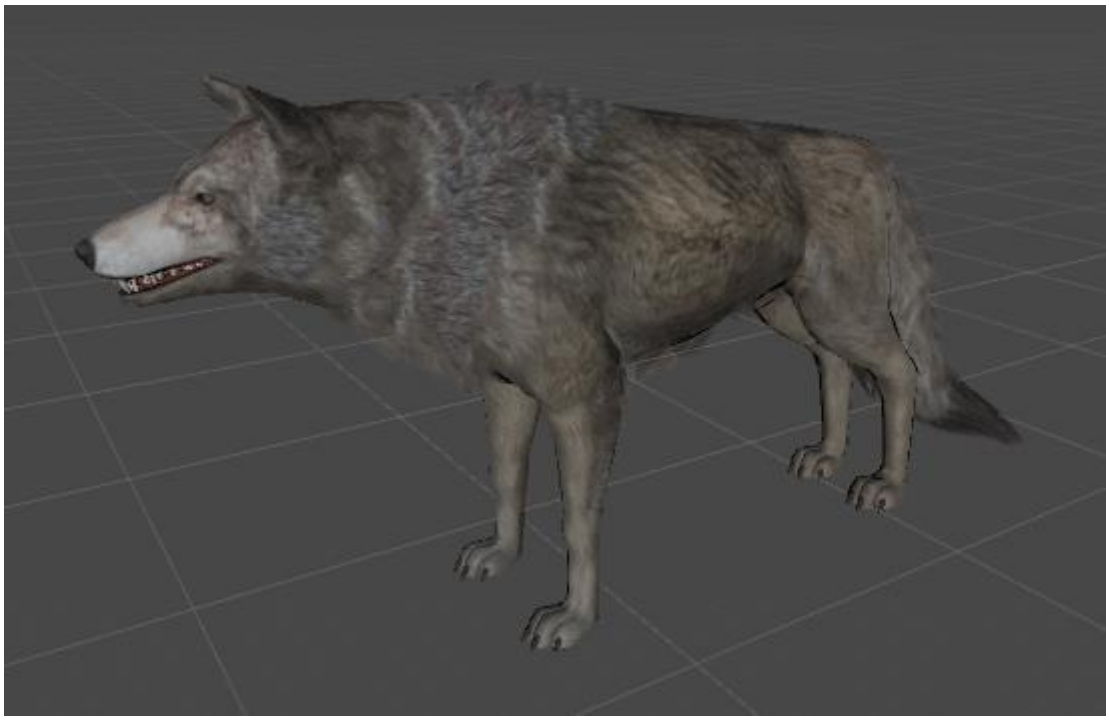


Ilustración 72. Modelo 3D de lobo

Para organizar todas las animaciones de un personaje he utilizado el *Animator Controller* [33] de Unity. Esta herramienta hace posible manejar diferentes estados de animaciones y transiciones entre estos, permitiendo definir parámetros para pasar de una animación a otra.

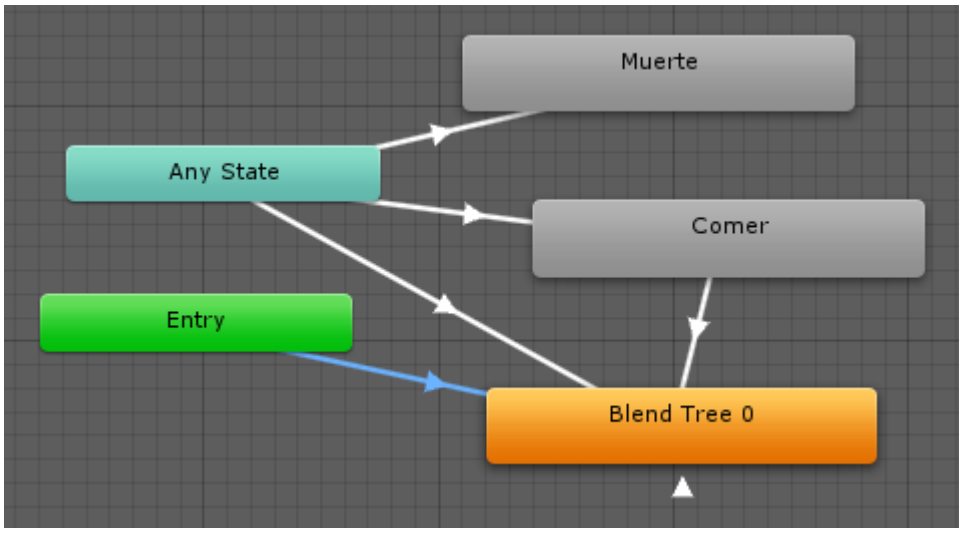


Ilustración 73. Animator Controller

Los animales disponen de diferentes animaciones: estado de reposo, desplazamiento, comer y morir.

Como se puede observar en la imagen anterior he utilizado un *Blend Tree* [34] para poder “combinar” las animaciones de reposo y desplazamiento y conseguir una mejor transición entre estas.

8.4.3. Interfaces

La pantalla de inicio consta de varios botones situados en el centro de la pantalla. Para todos los botones, al posar el cursor sobre ellos se oscurecerán.

Ilustración 74. Botones del menú de inicio

La pantalla de ajuste de parámetros tiene *sliders* para ajustar los valores, dos botones y un desplegable.

Ilustración 75. Pantalla de parámetros

Al pasar el cursor por encima de una etiqueta se mostrará información sobre esta.

Ilustración 76. Información adicional

Durante la ejecución de la partida se muestra en la parte superior derecha de la pantalla el tamaño de cada población, así como unas gráficas estadísticas que representan estos tamaños a lo largo del tiempo. Para crear las gráficas he utilizado el componente *Line Renderer* de Unity, que permite trabajar con líneas. Gracias a este componente puedo modificar la posición de los vértices de las líneas durante la ejecución, para así simular estas gráficas.

Ilustración 77. Tamaño poblaciones

Encima de cada animal se muestra un medidor que representa la energía del animal. Este se irá reduciendo según el animal pierde energía.

Ilustración 78. Barra de energía

El menú de pausa muestra varios botones: continuar, guardar, cargar y finalizar la partida y salir al menú principal.

Ilustración 79. Menú pausa

La pantalla de resultados muestra un diagrama de barras que compara el tamaño de las poblaciones iniciales con el que se tiene en el momento de finalizar la partida. También muestra el tiempo que ha durado la partida.

Ilustración 80. Poblaciones en la pantalla de resultados

Estas interfaces las he creado mediante el componente *Canvas* [35]. En un *Canvas* se añaden componentes de interacción, como botones y campos de texto, a los que se les asocia funcionalidad contenida en scripts.

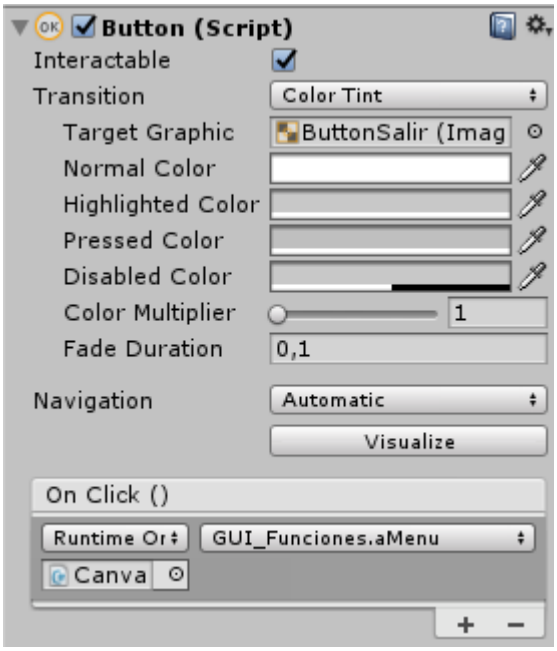


Ilustración 81. Inspector del componente Botón

8.4.4. Scripts

A continuación detallo los scripts creados para el proyecto:

- GameManager: es el controlador principal. Gestiona el estado del juego. También se encarga del proceso de guardado y carga de datos.
- AnimalControl: se encarga de gestionar las animaciones de los animales según su estado.
- CamaraControl: incluye la funcionalidad para el control de la cámara por parte del usuario.
- Generador: crea el entorno según los parámetros establecidos.
- Param: contiene la funcionalidad relativa a la interfaz de la pantalla de configuración de parámetros.
- Info: en este script utilizo eventos del cursor. Me ha permitido mostrar cuadros de texto con información al posar el cursor sobre una etiqueta de los parámetros.
- Gráficas: se encarga de crear las gráficas de la partida.

- GUI_Funciones: contiene el resto de funcionalidad relativa a la interfaz.
- BarraEnergía: controla el funcionamiento de la barra de energía de los animales.
- MenuPausa: script para la pausa del juego y la funcionalidad del menú.
- Generador: se encarga de la creación del entorno, redimensionando el terreno e instanciando los obstáculos, animales y plantas.
- Planta: gestiona la vida y regeneración de las plantas.
- Reproducción: script que se encarga del proceso de reproducción de los animales.
- Resultados: este script se encarga de generar la pantalla de resultados.
- PruebaAgent: es una clase derivada de la clase *Agent* de ML-Agents. Implementa la funcionalidad relativa a las observaciones, acciones y recompensas del agente.
- PruebaArea: contiene la funcionalidad relativa al área de entrenamiento.
- PruebaAcademy: subclase de *Academy* de ML-Agents.

Los scripts relativos a ML-Agents se encuentran en Assets/ML-Agents/Entrenamiento/Scripts. El resto está en Assets/Scripts.

8.4.5. Patrones de diseño

En el desarrollo de videojuegos hay algunos patrones de diseño especialmente útiles e indicados para resolver problemas particulares de este tipo de aplicaciones [36]. A continuación detallo los patrones de diseño aplicados en el proyecto:

- Patrón **Singleton**. El patrón Singleton lo he utilizado en el script *GameManager*, que se encarga de gestionar el estado actual del juego, y para esto hace falta que haya una única instancia de *GameManager* en todo momento.
- Patrón **Memento**. Lo he aplicado en el guardado de datos, al almacenar el estado de objetos, para restaurarlos posteriormente en la carga.
- Patrón **Prototipo**. Lo he utilizado en la generación del entorno. Teniendo distintos *prefabs* guardados, estos se instancian en el escenario.

8.4.6. Persistencia

En cualquier momento, el jugador podrá guardar el estado actual de la partida.

Para el almacenamiento y recuperación de la información realizo la serialización de objetos y guardo la información en un fichero llamado datos.dat.

8.4.7. Audio

Para añadir audio al juego utilizo el *Audio Source* de Unity. Mediante este se pueden gestionar clips de audio y ajustar diferentes parámetros del sonido.

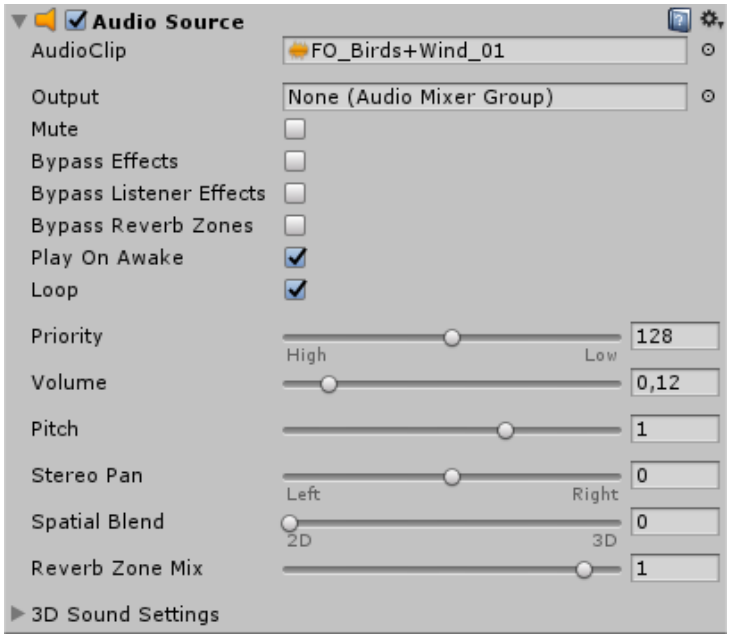


Ilustración 82. Audio Source de Unity

A continuación listo los elementos de audio que he utilizado para situaciones específicas:

- Sonido de fondo para el bosque
- Rugidos de lobo
- Audio de muerte
- Audio de comer
- Música de partida finalizada

8.4.8. Sistema de partículas

Para resaltar el nacimiento de un nuevo animal he añadido partículas. Para esto he utilizado el sistema de partículas de Unity [37], permite crear efectos muy variados a partir de una imagen. La imagen que he utilizado es la siguiente:

Ilustración 83. Imagen para partículas

Primero, es necesario crear un material a partir de una textura (la imagen mostrada) y un shader. He utilizado un shader específico para partículas que proporciona Unity.

Tras esto se crea un nuevo objeto de tipo *Particle System* al que se le proporciona el material creado y se ajustan sus características para crear el efecto deseado.

Finalmente, se lo he asociado como componente a cada animal. Las partículas se emitirán al generarse un nuevo animal.

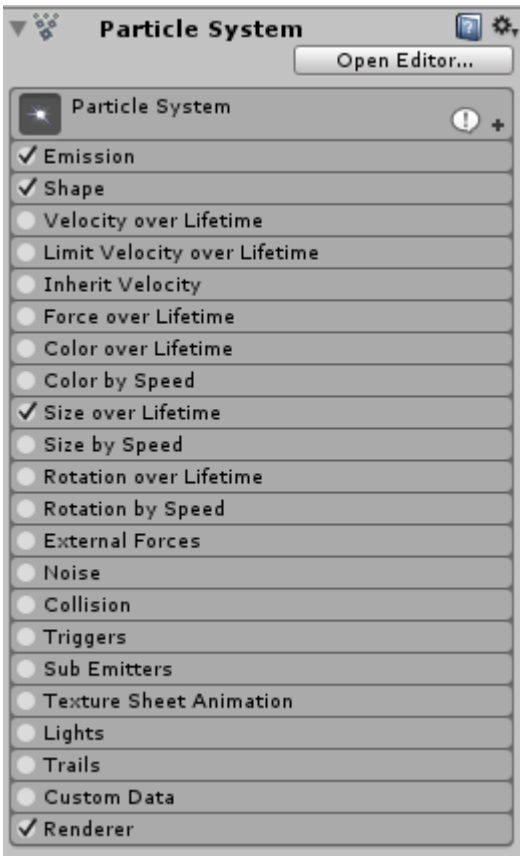


Ilustración 84. Particle System de Unity

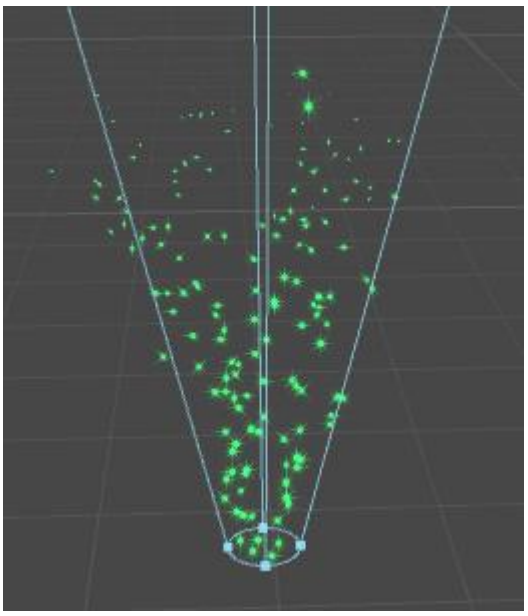


Ilustración 85. Partículas

8.4.9. Rendimiento

Para mejorar el rendimiento utilizo el occlusion culling de Unity. De esta forma se desactiva el renderizado de objetos cuando no sean visibles desde la cámara al estar tapados (occluded) por otros objetos.

Para esto, marco el terreno y objetos grandes como Occluder Static y Occludee Static y configuro los ajustes en la ventana de Occlusion Culling.

9. PRUEBAS

En este apartado se comprobará el correcto funcionamiento de la aplicación y el cumplimiento de los criterios de aceptación.

- **Desplazamiento**. Los animales se pueden mover por el entorno con una velocidad determinada.

Ilustración 86. Animal en movimiento

- **Comer**. Los animales carnívoros se alimentan de los herbívoros y estos últimos de plantas del entorno.

Ilustración 87. Lobo alimentándose de un conejo

- **Sistema de energía**. Los animales disponen de un sistema de energía. Al desplazarse consumen energía, así como al reproducirse, y la reponen alimentándose. Si se quedan sin energía mueren.

- **Muerte**. Un animal muere si se queda sin energía o es comido por un depredador.

Ilustración 88. Animal muerto por falta de energía

- **Visión**. Los animales no saben inicialmente la posición de su alimento. Conocen la posición de este una vez que ha entrado en su rango de visión.
- **Reproducción**. Los animales se pueden reproducir, generando descendencia a la que transmiten sus características.
- **Poblaciones**. Existen poblaciones de animales herbívoros y carnívoros. Mientras que los herbívoros se alimentan de plantas, los carnívoros se alimentan de estos primeros. Las plantas tienen un periodo regeneración tras ser comidas.
- **IA**. La inteligencia artificial de los animales ha sido creada con ML-Agents.
- **Obstáculos**. En el entorno hay árboles y terreno con los que pueden colisionar los animales.

Ilustración 89. Árbol con el que se puede colisionar

- **Parámetros**. El jugador puede indicar parámetros iniciales de las distintas poblaciones.
- **Pausa**. El jugador puede detener el juego en cualquier momento pulsando la tecla Escape.

Ilustración 90. Menú de pausa

- **Persistencia**. El jugador puede guardar el estado actual del juego y cargarlo posteriormente. Puede guardar y cargar desde el menú de pausa o cargar desde el menú principal. Si no existe partida guardada se muestra un mensaje de error.
- **Cámara**. El jugador puede mover y rotar la cámara con las teclas y el ratón respectivamente.
- **Entorno**. El tamaño del entorno es redimensionable. Su tamaño depende del número de animales que se haya establecido.
- **Representación**. Los elementos del juego están representados por modelos 3D, con animaciones y sonidos.
- **Interfaces**. El juego dispone de un menú de inicio, pantalla de configuración de ajustes y menú de pausa.
- **Información de juego**. Durante la partida se muestra información sobre las poblaciones.

Ilustración 91. Información durante la partida

- **Resultados**. Al finalizar el juego se muestra en la pantalla de resultados los cambios en las poblaciones.

10. EXPERIMENTACIÓN Y ANÁLISIS DE RESULTADOS

He realizado pruebas de la aplicación con diferentes parámetros para intentar conseguir un equilibrio de poblaciones.

Las características de los animales las he ajustado intentando que sean realistas. Los lobos se reproducen una vez al año [38] mientras que los conejos pueden llegar a tener hasta 7 partos al año por hembra [39], por lo que he establecido la frecuencia de reproducción de los animales en relación a esto. En cuanto a la velocidad, la de los lobos será ligeramente mayor, al ser estos animales más rápidos que los conejos [40]. El resto de las características son las mismas para las dos poblaciones de animales.

Tabla 22. Características establecidas

Por lo tanto, los parámetros que variarán en las pruebas son los tamaños de las poblaciones y el tipo de modelo.

A continuación se puede ver la tabla con los resultados:

Nº herbívoros	Nº Carnívoros	Nº Plantas	Modelo	Tiempo	Población extinta
8	8	8	10h	26s	Herbívoros
8	8	32	10h	2m16s	Herbívoros
8	32	8	10h	25s	Herbívoros
32	8	8	10h	>10m	-
32	32	8	10h	31s	Herbívoros
32	8	32	10h	>10m	-
8	32	32	10h	28s	Herbívoros
32	32	32	10h	1m51s	Herbívoros

8	8	8	8h	1m9s	Herbívoros
8	8	32	8h	20s	Herbívoros
8	32	8	8h	9s	Herbívoros
32	8	8	8h	>10m	-
32	32	8	8h	7m37s	Carnívoros
32	8	32	8h	>10m	-
8	32	32	8h	1m31s	Carnívoros
32	32	32	8h	8m33s	Carnívoros
8	8	8	5h	38s	Herbívoros
8	8	32	5h	26s	Herbívoros
8	32	8	5h	30s	Herbívoros
32	8	8	5h	>10m	-
32	32	8	5h	30s	Herbívoros
32	8	32	5h	>10m	-
8	32	32	5h	18s	Herbívoros
32	32	32	5h	8m21s	Carnívoros
8	8	8	3h	48s	Herbívoros
8	8	32	3h	56s	Herbívoros
8	32	8	3h	50s	Herbívoros
32	8	8	3h	1m47s	Herbívoros
32	32	8	3h	40s	Herbívoros
32	8	32	3h	1m34s	Herbívoros
8	32	32	3h	38s	Herbívoros
32	32	32	3h	51s	Herbívoros

Tabla 23. Resultados de diferentes partidas

Como se puede observar, las partidas que menos duran son aquellas con un número de herbívoros igual o menor que de carnívoros. Con estos parámetros una partida tarda alrededor de 2 minutos, al producirse la extinción de los conejos demasiado pronto.

Las partidas que han conseguido un equilibrio de poblaciones son aquellas que tienen un mayor número de herbívoros que de carnívoros. Esto permite que haya comida

para todos los carnívoros y que los herbívoros se puedan reproducir de manera estable.

Tener un número adecuado de plantas también es importante, para que los herbívoros no se mueran de hambre.

En cuanto al modelo, el de 10 horas (en el entrenamiento de este modelo se obtuvo una recompensa media de 0,94) es el que ha demostrado el mejor comportamiento. Con el de 8 h (recompensa media de 0,87) el comportamiento es muy parecido al de 10 h. Esto es porque la recompensa media de ambos es muy similar. Al cambiar al de 5 h (recompensa media de -4.35) empeora el movimiento de los agentes, al mantenerse en las mismas zonas durante mucho tiempo. Cambiando al de 3 h (que produjo una recompensa media final de -9.56) se obtiene un comportamiento no adecuado. Los agentes se desplazan moviéndose continuamente en direcciones contrarias y muy lentamente, por lo que no llegan a obtener comida y suelen morirse por falta de energía.

Los casos en los que se han extinguido los lobos se debe al uso de un modelo con un comportamiento no adecuado.

11. CONCLUSIONES Y DESARROLLOS FUTUROS

11.1. Conclusiones

Tras haber utilizado ML-Agents en este proyecto he comprobado que se trata de una herramienta que, al igual que Unity, tiene una gran cantidad de aplicaciones.

ML-Agents es muy recomendable para el desarrollo de comportamiento inteligente en agentes, especialmente si se lleva a cabo en entornos totalmente observables.

Aunque antes de utilizarlo en un proyecto es necesario valorar, dependiendo de la complejidad y el tipo de tarea, si puede ser más rentable programar a medida la inteligencia artificial de un agente. Utilizando ML-Agents, obtener el comportamiento deseado suele requerir muchas pruebas y, dependiendo de los parámetros establecidos, el entrenamiento puede llevar mucho tiempo. Sin embargo, aprendiendo a utilizar la herramienta, implementando un buen entorno de entrenamiento y ajustando los parámetros adecuados se pueden obtener muy buenos resultados.

En el entrenamiento, las recompensas positivas son más útiles que las negativas para formar el comportamiento deseado del agente. Sin embargo, los agentes se pueden “aprovechar” de las recompensas positivas. Es decir, estos pueden reforzar un tipo de comportamiento del que obtienen recompensa positiva y no llegar a conseguir el objetivo deseado.

En cuanto a las recompensas negativas, si son demasiado grandes pueden producir un comportamiento no deseado en el que los agentes evitan cualquier tipo de comportamiento que pudiera llegar a producir recompensa negativa, incluso aunque finalmente llegaran a obtener recompensa positiva.

Conforme aumenta el tamaño del problema se complica el entrenamiento. Por ello es bueno ir añadiendo poco a poco nueva funcionalidad e ir comprobando el comportamiento obtenido en los entrenamientos.

También es muy importante ajustar los parámetros del algoritmo PPO teniendo en cuenta el tipo de proyecto y observar las gráficas del entrenamiento para saber si los parámetros son los adecuados para obtener buenos resultados.

Por otro lado, se puede mejorar la eficacia paralelizando el entrenamiento, al duplicar las áreas de entrenamiento o utilizar más de un agente con el mismo cerebro.

Por lo tanto, ML-Agents resulta una herramienta muy útil en el desarrollo de videojuegos. Aunque no solo para esto, ML-Agents se puede utilizar en otros tipos de tareas, y se pueden beneficiar diferentes grupos. Desarrolladores de videojuegos pueden crear agentes inteligentes con comportamiento dinámico. Investigadores de inteligencia artificial pueden estudiar el comportamiento de múltiples agentes en escenarios realistas de diferente tipo. Mientras que investigadores industriales pueden implementar sistemas de entrenamiento para robótica, vehículos autónomos y otras aplicaciones industriales.

En cuanto a Unity, es un motor de juegos que facilita en gran medida el desarrollo de proyectos por su interfaz intuitiva y amigable, la integración de diferentes herramientas y por disponer de un extenso manual online. Además, dispone de una amplia comunidad de usuarios, lo que permite encontrar rápidamente soluciones y ayuda a posibles dudas o problemas.

Personalmente, el desarrollo de este proyecto me ha servido para aplicar conocimientos obtenidos en el máster y el grado y adquirir nuevos. He utilizado con éxito una metodología ágil, he podido aprender sobre *machine learning* y aplicarlo a un proyecto. Además, he seguido adquiriendo conocimiento sobre el desarrollo de aplicaciones en Unity, he aprendido a utilizar la herramienta ML-Agents y he conseguido el comportamiento inteligente que necesitaba para este proyecto.

11.2. Posibles desarrollos futuros

En el momento del desarrollo de este proyecto ML-Agents se encuentra en versión beta, por lo que una posible mejora del proyecto sería actualizarlo a una futura versión final.

Otra posible mejora del proyecto podría ser añadir poblaciones de animales de otras especies.

12. ANEXOS

12.1. Glosario

Glosario ML-Agents:

- **Academia**: componente de Unity que controla los ajustes de temporización, reinicio y entrenamiento/inferencia del entorno.
- **Acción**: la toma de una decisión por parte de un agente dentro del entorno.
- **Agente**: componente de Unity que produce observaciones y realiza acciones en el entorno. Las acciones de los agentes están determinadas por las decisiones tomadas por un cerebro vinculado.
- **Cerebro**: componente de Unity que toma decisiones para los agentes vinculados a este.
- **Decisión**: la especificación producida por un cerebro para que se realice una acción dada una observación.
- **Editor**: el Editor de Unity, que puede incluir cualquier panel (por ejemplo, Jerarquía, Escena, Inspector).
- **Entorno**: la escena de Unity que contiene agentes, academia y cerebros.
- **Observación**: información parcial que describe el estado del entorno disponible para un agente determinado.
- **Política**: función para producir decisiones a partir de observaciones.
- **Recompensa**: señal proporcionada en cada paso utilizada para indicar la conveniencia de la acción de un agente dentro del estado actual del entorno.
- **Estado**: las propiedades del entorno (incluidos todos los agentes dentro de él) en un momento dado.
- **Paso**: corresponde a cada llamada FixedUpdate del motor de juego. Es la menor tasa de cambio posible en el estado.
- **FixedUpdate**: método de Unity llamado en cada actualización del motor de juego.
- **Update**: función de Unity llamada cada vez que se procesa un fotograma.

- **Comunicador externo**: clase de ML-Agents responsable de la comunicación con procesos externos (en este caso, la API de Python).
- **TensorFlow**: biblioteca de software de código abierto para el cálculo numérico que utiliza diagramas de flujo de datos. Los nodos de los grafos representan operaciones matemáticas, mientras que las aristas representan los conjuntos de datos multidimensionales (tensores) que fluyen entre ellos.

Glosario Unity [41]:

- **Escena**: fichero que contiene los entornos o menús de un juego. Cada escena se puede entender como un “nivel” de un juego.
- **Asset**: representación de cualquier elemento que puede usarse en un proyecto. Un asset puede provenir de un archivo creado fuera de Unity, como un modelo 3D, un archivo de audio, una imagen u otros tipos de archivos que admita Unity. También hay algunos tipos de assets que se pueden crear dentro de Unity, como un *Animator Controller*.
- **GameObject**: objetos esenciales en Unity que representan personajes, decorado o el escenario. Funcionan como contenedores para componentes que implementan la funcionalidad.
- **Prefab**: plantilla a partir de la cual se pueden crear nuevas instancias de un *GameObject* en la escena.
- **Terrain**: herramienta de Unity para crear terrenos. Permite editar su altura y añadirle texturas, hierba y árboles.
- **Skybox**: envoltura geométrica que se renderiza alrededor de la escena para simular el cielo.
- **Animator Controller**: herramienta que permite organizar y mantener un conjunto de clips de animación y transiciones de animación asociadas a un personaje u objeto.

12.2. Manual de la aplicación

Al iniciar la aplicación se muestra el menú principal. Este dispone de varios botones:

- **Nueva partida**, para comenzar una nueva partida estableciendo parámetros.
- **Cargar partida**; carga una partida previamente guardada.
- **Salir**, para cerrar la aplicación.

Al seleccionar **Nueva partida** aparecerá la pantalla de configuración de parámetros. Aquí podrá ajustar:

- Para los animales:
  - El tamaño de la población
  - La velocidad de movimiento
  - El rango de visión
  - La frecuencia de reproducción
  - La cantidad de energía
- Para las plantas solo el tamaño de su población.
- El modelo entrenado a utilizar en el cerebro de los agentes. Se diferencian en el tiempo que ha durado su entrenamiento, por lo que los de mayor tiempo resultan en una recompensa media más alta y tendrán un mejor comportamiento:
  - De 10 horas. En el entrenamiento de este modelo se obtuvo una recompensa media de 0,94.
  - De 8 horas. Con una recompensa media de 0,87.
  - De 5 horas. Con una recompensa media de -4,35.
  - De 3 horas. Con una recompensa media de -9,56

Una vez editado lo deseado se puede iniciar la partida pulsando en **Comenzar**.

Tras esto se dirigirá al entorno 3D donde los agentes llevarán a cabo su comportamiento.

Utilice el **ratón** para mover la cámara y las teclas **W**, **A**, **S**, **D** para desplazarse por el entorno. También puede moverse en vertical utilizando **Q** para bajar y **E** para subir. También puede aumentar la velocidad de movimiento presionando **Shift**.

En la parte superior derecha de la pantalla aparece información sobre el tamaño de las poblaciones: **H** indica los herbívoros, **C** los carnívoros y **Pl** las plantas.

Pulse **ESC** para activar la pausa. Aquí se puede elegir continuar la partida, guardarla, cargar una previamente guardada o salir al menú principal. Para continuar la partida también se puede volver a pulsar **ESC**.

Puede aumentar la velocidad a la que se desarrolla el juego pulsando la tecla **R**.

Al finalizar la partida se mostrará la pantalla de resultados. La partida finalizará si alguna de las poblaciones se extingue o si se pulsa el botón **Finalizar** en el menú de pausa.

En la pantalla de resultados se mostrará un diagrama de barras que compara el tamaño inicial de las poblaciones con el tamaño final. Para volver al menú principal pulse **Salir**.

12.3. Contenido del DVD

En el DVD se puede encontrar lo siguiente:

- Vídeo explicativo de la aplicación.
- Ejecutable de la aplicación para PC.
- Proyecto de Unity. Los scripts creados se pueden encontrar en Assets/Scripts y en Assets/ML-Agents/Entrenamiento/Scripts.
- Esta memoria.

BIBLIOGRAFÍA

[1] A. Juliani, «Introducing: Unity Machine Learning Agents,» septiembre 2017. [En línea]. Available: https://blogs.unity3d.com/es/2017/09/19/introducing-unity-machine-learning-agents/.

[2] «Unity ML-Agents (Beta),» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents.

[3] «Machine Learning,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Background-Machine-Learning.md. [Último acceso: Marzo 2018].

[4] «Using Machine Learning Agents in a real game,» [En línea]. Available: https://blogs.unity3d.com/es/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/. [Último acceso: Febrero 2018].

[5] «Unity,» [En línea]. Available: https://unity3d.com/es. [Último acceso: 2018].

[6] «Visual Paradigm,» [En línea]. Available: https://www.visual-paradigm.com/. [Último acceso: 2018].

[7] «XMind,» [En línea]. Available: http://www.xmind.net/. [Último acceso: 2018].

[8] «Incremental model- advantages, disadvantages and when to use it,» [En línea]. Available: http://istqbexamcertification.com/what-is-incremental-model-advantages-disadvantages-and-when-to-use-it/. [Último acceso: 2018].

[9] «Principles behind the Agile Manifesto,» [En línea]. Available: http://agilemanifesto.org/principles.html. [Último acceso: 2018].

[10] «The iron triangle of planning. Atlassian,» [En línea]. Available: https://www.atlassian.com/agile/agile-at-scale/agile-iron-triangle.

[11] D. Green, Scrum: Novice to Ninja, 2016.

[12] «The Scrum Guide,» [En línea]. Available: https://www.scrumalliance.org/learn-about-scrum/the-scrum-guide. [Último acceso: 2018].

[13] «Agile Story Card Templates,» [En línea]. Available: https://www.solutionsiq.com/learning/blog-post/agile-story-card-templates/. [Último acceso: 2018].

[14] «MoSCoW prioritization of the product backlog,» [En línea]. Available: https://www.scrumdesk.com/start/manual-for-scrumdesk-start/start-moscow-prioritization-product-backlog/. [Último acceso: 2018].

[15] «Salario bruto mensual para: Ingeniero de software informático,» [En línea]. Available: https://tusalario.org/usa/main/salario/compara-tu-salario?job-id=2512010000000#/. [Último acceso: 2018].

[16] K. Pohl y C. Rupp, «Three Perspectives on the Requirements,» de *Requirements Engineering Fundamentals*, 2011.

[17] E. Adams y J. Dormans, Game Mechanics: Advanced Game Design, 2012.

[18] «Designing with Gameplay Modes and Flowboards,» [En línea]. Available: https://www.gamasutra.com/view/feature/130514/designers_notebook_designing_.php. [Último acceso: Marzo 2018].

[19] «ML-Agents Overview,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md. [Último acceso: Febrero 2018].

[20] «Dota 2 - OpenAI,» [En línea]. Available: https://blog.openai.com/dota-2/. [Último acceso: 2018].

[21] «Training with Proximal Policy Optimization,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md. [Último acceso: Marzo 2018].

[22] «Proximal Policy Optimization,» [En línea]. Available: https://blog.openai.com/openai-baselines-ppo/. [Último acceso: Marzo 2018].

[23] « Proximal Policy Optimization Algorithms,» [En línea]. Available: https://arxiv.org/abs/1707.06347. [Último acceso: Marzo 2018].

[24] J. Peters, «Policy gradient methods,» [En línea]. Available: http://www.scholarpedia.org/article/Policy_gradient_methods.

[25] « High-Dimensional Continuous Control Using Generalized Advantage Estimation,» [En línea]. Available: https://arxiv.org/abs/1506.02438. [Último acceso: Marzo 2018].

[26] «ML-Agents Installation & Set-up,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Installation.md. [Último acceso: Enero 2018].

[27] «Making a New Learning Environment,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md. [Último acceso: Febrero 2018].

[28] «Reinforcement Learning in Unity,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design.md. [Último acceso: Marzo 2018].

[29] «Environment Design Best Practices,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Best-Practices.md. [Último acceso: Febrero 2018].

[30] «Getting Started with the 3D Balance Ball Environment,» [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Getting-Started-with-Balance-Ball.md. [Último acceso: Febrero 2018].

[31] «Unity Terrain,» [En línea]. Available: https://docs.unity3d.com/Manual/terrain-UsingTerrains.html. [Último acceso: Marzo 2018].

[32] «Unity Skybox,» [En línea]. Available: https://docs.unity3d.com/Manual/class-Skybox.html. [Último acceso: Marzo 2018].

[33] «Unity Animator Controller,» [En línea]. Available: https://docs.unity3d.com/Manual/Animator.html. [Último acceso: Marzo 2018].

[34] «Unity Blend Trees,» [En línea]. Available: https://docs.unity3d.com/Manual/class-BlendTree.html. [Último acceso: Marzo 2018].

[35] «Canvas - Unity User Manual,» [En línea]. Available: https://docs.unity3d.com/Manual/UICanvas.html. [Último acceso: 2018].

[36] «Game Programming Patterns,» [En línea]. Available: http://gameprogrammingpatterns.com/contents.html. [Último acceso: Marzo 2018].

[37] «Using Particle Systems in Unity,» [En línea]. Available: https://docs.unity3d.com/Manual/PartSysUsage.html. [Último acceso: 2018].

[38] «Reproducción de los lobos,» [En línea]. Available: https://www.wolfworlds.com/es/reproduccion-de-los-lobos/. [Último acceso: 2018].

[39] «Reproducción del conejo,» [En línea]. Available: https://www.botanical-online.com/animales/reproduccion_conejo.htm. [Último acceso: 2018].

[40] «Velocidad de los animales,» [En línea]. Available: https://es.wikipedia.org/wiki/Velocidad_de_los_animales. [Último acceso: 2018].

[41] «Manual Unity,» [En línea]. Available: https://docs.unity3d.com/Manual/UnityManual.html. [Último acceso: Marzo 2018].

[42] «An introduction to personas and how to create them,» [En línea]. Available: http://www.steptwo.com.au/papers/kmc_personas/. [Último acceso: 2018].

[43] «ML-Agents, Página principal,» [En línea]. Available: https://unity3d.com/es/machine-learning/. [Último acceso: 2018].