UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Redes Neuronales aplicadas a un videojuego en el entorno de Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Noguera Guijarro, Francisco José

Tutor/a: Mollá Vayá, Ramón Pascual

CURSO ACADÉMICO: 2023/2024

Resum

Aquest projecte tindrà com a objectiu principal l'anàlisi i la comprensió del funcionament de les xarxes neuronals des d'un enfocament més pràctic. Aixína, a partir de la idea d'un videojoc simple que puga explotar les xarxes neuronals, es pretén aprendre de primera mà com implementar i usar les mateixes a l'entorn de Unity. Per tant, el treball estarà compost principalment per la documentació del desenvolupament del propi videojoc, on s'explicarà el concepte del joc, com s'ha aconseguit tot l'apartat artístic, com serà la jugabilitat..., i per la investigació i estudi de les xarxes neuronals en aquest camp. Per a això últim, es buscaran i utilitzaran diferents tecnologies que permeten aconseguir aquesta comesa, com pot ser ML -Agents, i es realitzarà una comparativa entre aquestes amb diverses mètriques amb la finalitat d'obtindre conclusions rellevants.

Paraules clau: Xarxes Neuronals, Intel·ligència Artificial, Videojocs, Unity, Programació, Disseny de videojocs

Resumen

Este proyecto tendrá como objetivo principal el análisis y la comprensión del funcionamiento de las redes neuronales desde un enfoque más práctico. Así, a partir de la idea de un videojuego simple que pueda explotar las redes neuronales, se pretende aprender de primera mano cómo implementar y usar las mismas en el entorno de Unity. Por tanto, el trabajo estará compuesto principalmente por la documentación del desarrollo del propio videojuego, donde se explicará el concepto del juego, como se ha conseguido todo del apartado artístico, como será la jugabilidad..., y por la investigación y estudio de las redes neuronales en este campo. Para esto último, se buscarán y utilizarán diferentes tecnologías que permitan conseguir este cometido, como puede ser ML Agents, y se realizará una comparativa entre estas con diversas métricas con el fin de obtener conclusiones relevantes.

Palabras clave: Redes Neuronales, Inteligencia Artificial, Videojuegos, Unity, Programación, Diseño de videojuegos

Abstract

This project will have as main objective the analysis and understanding of the functioning of the neural networks from a more practical approach. Thus, from the idea of a simple video game that can exploit neural networks, it is intended to learn first-hand how to implement and use them in the Unity environment. Therefore, the project will be composed mainly of the documentation of the development of the video game itself, where the concept of the game will be explained, how everything has been achieved in the artistic section, how the gameplay will be..., and of the research and study of neural networks in this field. To achieve this last thing, different technologies will be sought and used to fulfil this task, such as ML Agents, and a comparison will be made between these with various metrics in order to obtain relevant conclusions.

Key words: Neural Network, Artificial Intelligence, Video games, Unity, Programming, Game Design

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En este primer capítulo introductorio se verán diversos apartados que, por un lado, justifican y dan sentido a la existencia de este trabajo, y por otro lado, muestran las bases y principios a partir de los cuáles se va a ir construyendo todo lo demás.

1.1 Motivación

Desde muy joven me ha fascinado el mundo de los videojuegos, y esta es la razón principal por la que he decidido enfocar el desarrollo de mi TFG alrededor del diseño y la creación de uno. Además, este pensamiento ha sido fomentado gracias a que en mi último año de carrera he tenido la suerte de poder participar en dos asignaturas que me han permitido vivir de primera mano, aunque en pequeña escala, como es trabajar en un equipo de desarrollo en este sector, donde incluso uno fue en un ámbito multidisciplinar. Esto también me ha ayudado a decantarme por usar el motor de videojuegos de Unity.

Una vez con esto en mente, faltaba tratar alguna cosa que relacionase este proyecto con lo estudiado en la rama de computación, y algo que parecía bastante apropiado, siendo de gran relevancia en los videojuegos, era el apartado de la inteligencia artificial. Alrededor de este tema se pueden trabajar a partir de distintas bases, como máquinas de estado o árboles de decisión, sin embargo, aquí entra otro apartado que también me ha llamado mucho la atención y que me gustaría entender mejor como funciona: las redes neuronales.

Así, con todo esto ya decidido, han ido surgiendo las distintas ideas y conceptos que han llevado a que este trabajo haya sido posible.

1.2 Objetivos

Este proyecto tiene como objetivo principal entender más profundamente el funcionamiento de las redes neuronales, así como estudiar, en este caso, el funcionamiento de algunas alternativas de implementación en el entorno de Unity. Para ello, y teniendo en cuenta que la idea es alcanzarlo a través de la elaboración de un videojuego, se realizará los siguiente subobjetivos:

1. Diseñar un videojuego funcional desde cero.
2. Buscar y analizar distintas opciones de integración de redes neuronales a Unity, entre las que se encuentren *ML-Agents* junto con alguna adicional.
3. Implementar y entrenar una red neuronal con cada opción, así como realizar los ajustes y cambios que puedan requerir particularmente cada una.
4. Obtener conclusiones a partir de los resultados obtenidos que nos permitan valorar costes de entrenamiento, esfuerzo necesario y calidad final del modelo.

1.3 Metodología

Principalmente se usará el método Kanban para gestionar lo mejor posible el avance del proyecto. Este sistema se basa en el uso de tarjetas, en nuestro caso digitales a partir de la plataforma de Trello, que permitan seguir de forma visual el flujo de trabajo.

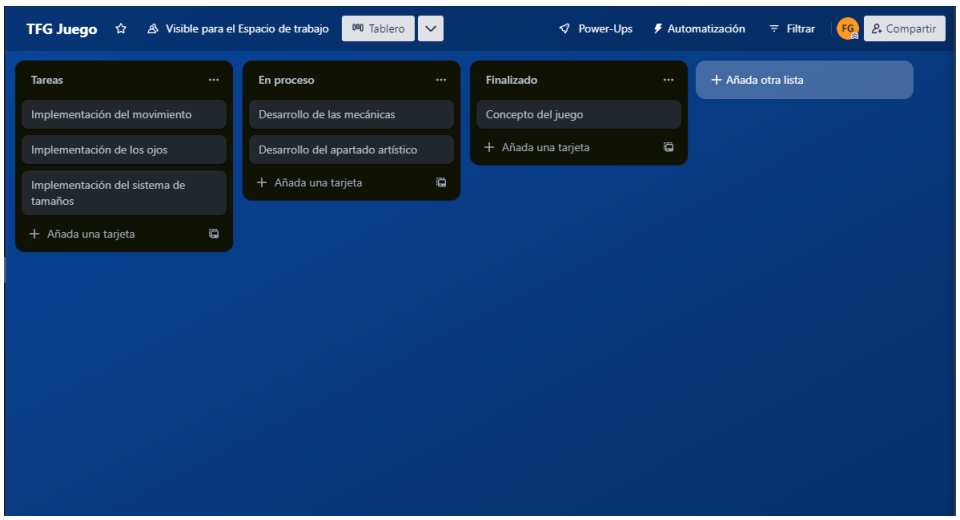


Figura 1.1: Tabla de trello al comienzo del desarrollo del videojuego

En la figura 1.1 se puede observar las columnas que se utilizarán para determinar el estado de una tarea:

- Tareas: Aquí aparecerán las recién creadas.
- En proceso: Se muestran aquellas en las que este trabajando en ese momento.
- Finalizado: Se encontrarán las ya acabadas.

Además, se usarán dos tablas para distinguir, por un lado, la parte de diseñar y crear el propio videojuego, y por otro, el incorporar e implementar todo el apartado de las redes neuronales. Con todo esto se busca mantener cierta organización que ayudé tanto en controlar y conseguir lo objetivos marcados, como en facilitar la posterior documentación de lo realizado.

Por otro lado, se ha elaborado un Diagrama de Gantt para contribuir al control de las tareas de una forma más global siendo la figura 1.2 el resultado inicial.

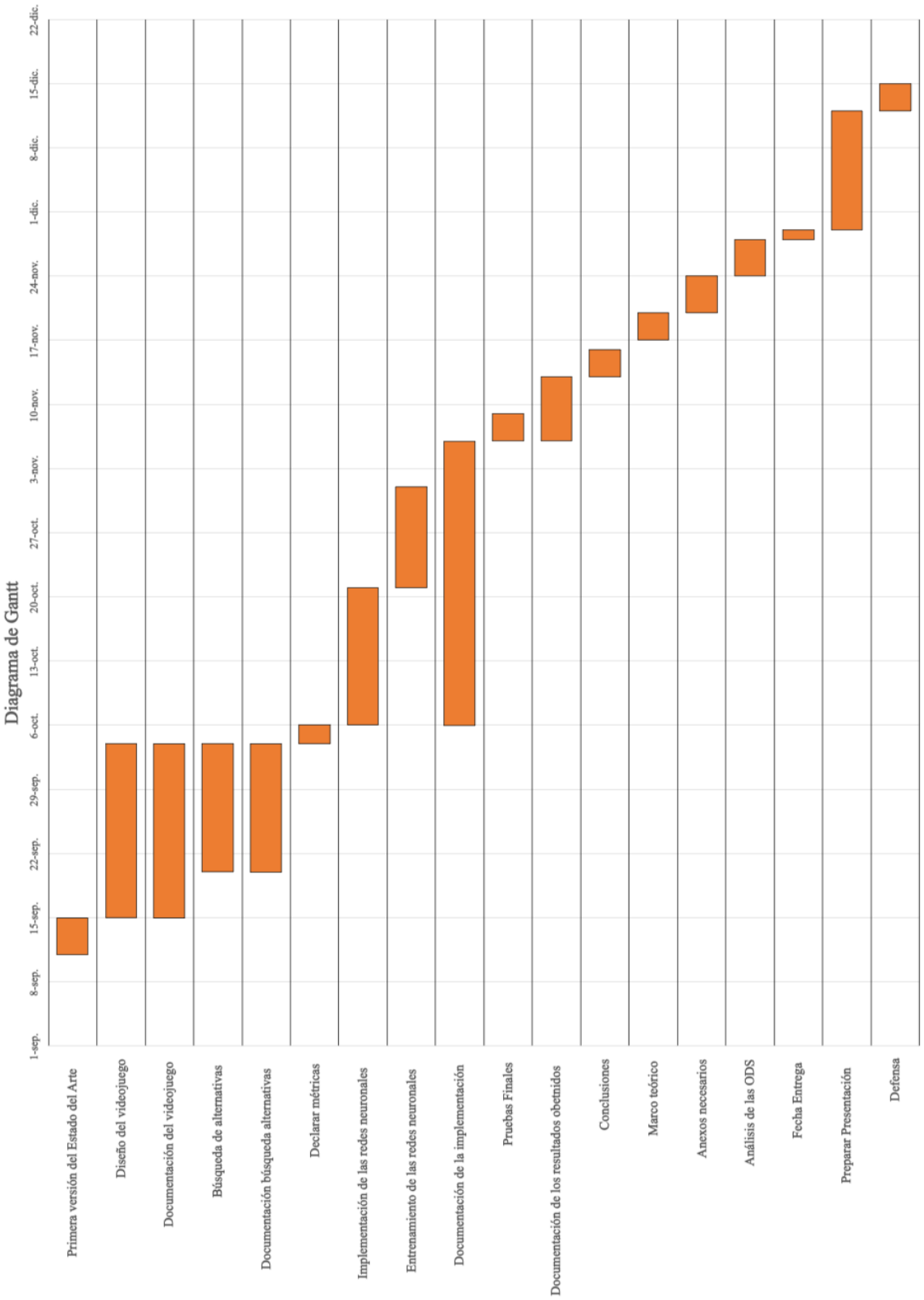


Figura 1.2: Diagrama de Gantt al inicio del proyecto

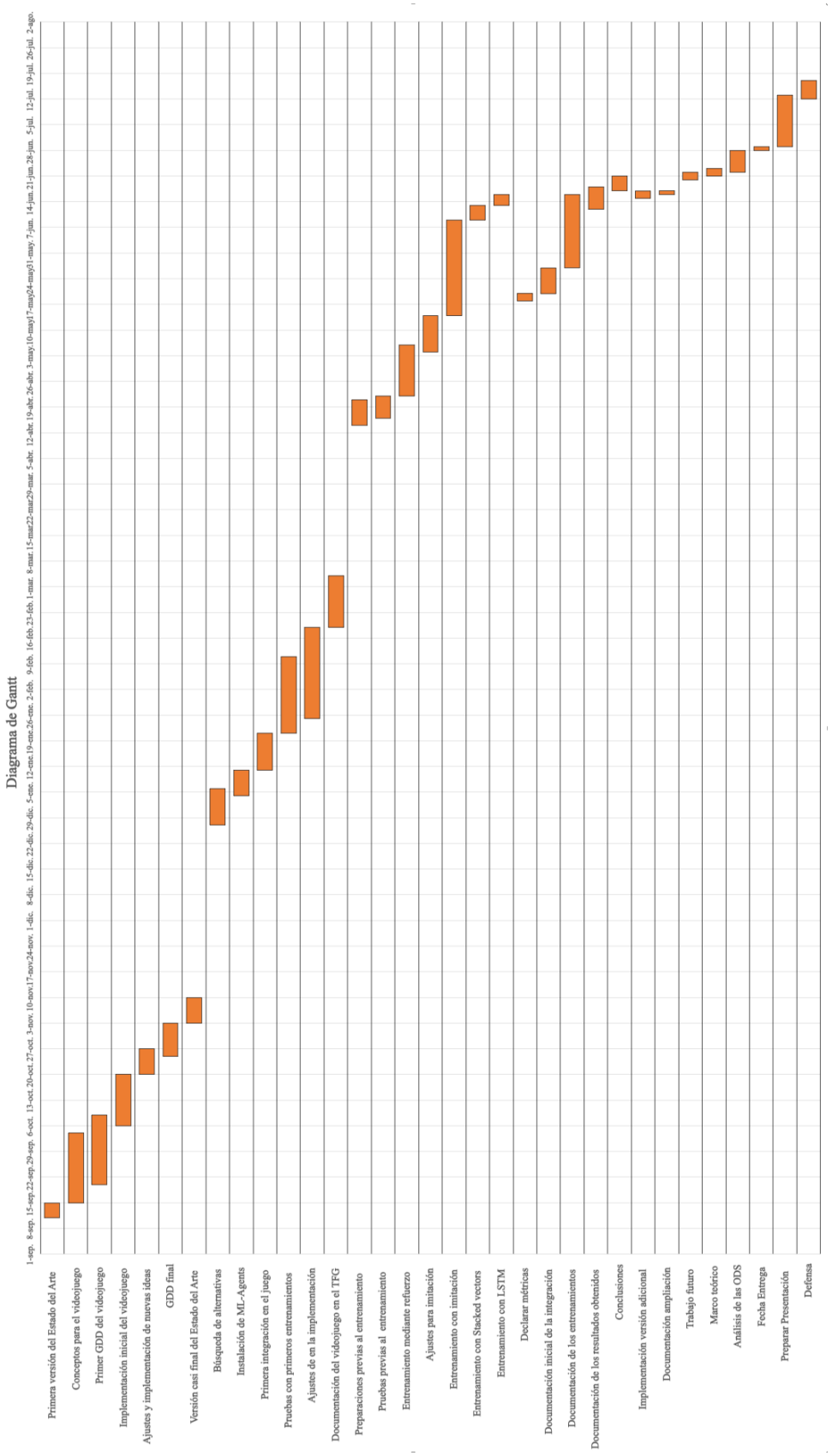


Figura 1.3: Diagrama de Gantt al final del proyecto.

Sin embargo, al final del proyecto, como se puede ver en la figura 1.3, han sido necesario ajustes debido a la aparición de elementos no valorados al inicio como pausas en el desarrollo debidas a descansos prolongados durante un tiempo excesivo que se han

mezclado con asuntos personales, o partes que requieren de más pasos. Como se puede observar en relación a las tareas que se plantearon al comienzo, han terminado desglosadas y extendidas en tareas más concretas, ocurriendo casos como la implementación o entrenamiento de las redes, donde han pasado a tres o más subtareas. También, debido al desarrollo de la documentación, al final ha habido partes que se han extendido más de lo necesario con más solapamientos y tareas en paralelo.

1.4 Estructura de la memoria

Para esta memoria se intentará mantener una estructura que permita al lector seguir la secuencia de pasos realizada durante el desarrollo del proyecto. Así, en primer lugar, se tiene este apartado de introducción donde, como ya se ha mencionado, se justifica la idea de este TFG, se presentan los objetivos que se pretenden alcanzar, cual ha sido mi motivación inicial, cuál será la metodología a utilizar y se presentarán la herramientas principales utilizadas.

A continuación se verá el estado del arte en el capítulo 2, donde primeramente se verá la situación actual de las tecnologías y campos que se encuentren relacionados con los temas que se tratarán. En concreto, de los motores de videojuegos, de las distintas herramientas de edición de imágenes digitales, de los entornos de programación y finalmente una apartado para la inteligencia artificial en los videojuegos.

Luego, en el capítulo 3, se darán los fundamentos teóricos necesarios que permitan entender lo mejor posible los contenidos de este documento.

Seguidamente se comenzará el capítulo 4 donde se desenvuelve el diseño y desarrollo del videojuego. Esta parte se dividirá en diversas secciones donde se irá explicando detalladamente como se han elaborado los distintos elementos del juego. En concreto, al haberse realizado en Unity, se hará una distinción entre escenas, y dentro de cada una se explicarán sus componentes más importantes.

Posteriormente se tratará lo que es posiblemente lo más importante e interesante de este proyecto: la implementación e incorporación de las redes neuronales al videojuego realizado en el capítulo 5. Uno de los aspectos clave es que se va a intentar conseguir esto a través de diversas alternativas que se encuentren en la web, y de igual forma, este capítulo se dividirá de tal forma que ayude a ver el orden de sucesos que ha llevado a obtener los resultados finales.

Para acabar con lo que es el desarrollo del trabajo, se realizará unos análisis de estos resultados en el capítulo 6. Con esto se pretende comparar y comprender mejor que aportan las distintas opciones barajadas en el apartado anterior.

Antes de terminar, habrá un pequeño apartado en el capítulo 7 donde se detallará el desarrollo de un elemento que no afecta a los resultados del proyecto.

Finalmente el apartado de conclusiones en el capítulo 8, donde se refleje cual ha sido mi experiencia y opinión final, así como un sumario de todo lo obtenido como resultado.

Como apartado adicional, en el capítulo 9 se detallarán un conjunto de posibilidades que permitirían ampliar diversas partes el trabajo.

1.5 Herramientas utilizadas

A continuación se presentarán las diversas herramientas utilizadas a lo largo de la creación de este trabajo:

- Unity[1]: Motor de videojuegos desarrollado por Unity Technologies. Aunque existen otras alternativas también gratuitas, para este trabajo lo que proporciona es más que suficiente.

Figura 1.4: Logo de Unity

- GIMP[2]: Programa de edición de imágenes digitales. Utilizado principalmente para tratar todo lo necesario del apartado artístico durante el desarrollo del videojuego.

Figura 1.5: Logo de GIMP

- Visual Studio 2022[3]: Un IDE para programar y depurar en distintos lenguajes de programación. En nuestro caso es interesante por su gran integración con el motor de Unity, donde usaremos C#.

Figura 1.6: Logo de Visual Studio

[1]Página web oficial de Unity: https://unity.com/es
[2]Página web oficial de GIMP: http://www.gimp.org.es/
[3]Página web oficial de VS2022: https://visualstudio.microsoft.com/es/vs/

- Github[4]: Es una plataforma para almacenar proyectos, y se usará precisamente para mantener en la nube un repositorio con todo lo realizado en el proyecto. Además, esto también permite que otros usuarios puedan tener acceso al trabajo fácilmente.

Figura 1.7: Logo de Github

- Trello[5]: Se trata de un software para administrar proyectos. Como se comentó más en detalle en el apartado de metodología, se utilizará para controlar el correcto desarrollo de las distintas partes que forman este trabajo.

Figura 1.8: Logo de Trello

- Overleaf[6]: Para el desarrollo de la memoria se hará con el sistema LaTeX, siendo Overleaf un editor online gratuito para escribir en este sistema.

Figura 1.9: Logo de Overleaf

[4]Página web oficial de Github: https://github.com/
[5]Página web oficial de Trello: https://trello.com/es
[6]Página web de Overleaf: https://es.overleaf.com/

CAPÍTULO 2
Estado del arte

En esta sección, como ya se ha explicado anteriormente, se verá la situación actual de las tecnologías y herramientas más relevantes para este proyecto, siendo los motores de videojuegos, las herramientas de edición de imágenes digitales, los entornos de programación y la inteligencia artificial los apartados a tratar.

2.1 Motores de videojuegos

Un motor de videojuegos es un software que proporciona un entorno compuesto de diversos componentes enfocados y preparados para la creación de videojuegos. Así, entre las herramientas más importantes que pueden proporcionar se encuentran:

- Un apartado gráfico encargado de la renderización en pantalla
- Un motor de físicas
- Herramientas para controlar la entrada/salida
- Un componente encargado de la gestión del audio
- Herramientas para controlar la inteligencia artificial
- Componentes para dar soporte de red
- Sistema de scripting

El motor esta creado a partir de unificar e integrar todas estas herramientas con el fin de facilitar y simplificar la tarea de crear videojuegos lo máximo posible. Esto se consigue a través de eliminar las complejas tareas técnicas necesarias que implicaría tener que desarrollar cada uno de los distintos componentes, permitiendo al usuario explotar la parte más creativa de la creación de videojuegos. Para más detalles sobre este tema visitar [1]. Una vez visto por encima que son los motores de videojuegos, se puede pasar a ver cuales son actualmente los más usados e importantes del mercado, además de sus características más relevantes.

2.1.1. Unreal Engine 5

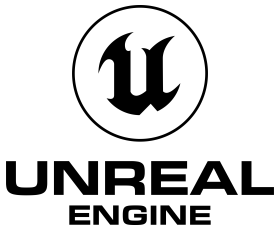


Figura 2.1: Logo de Unreal Engine

Unreal Engine 5, desarrollado por Epic Games, es uno de los motores más populares actualmente y, siendo precursor del UE4, ofrece nuevas tecnologías que permiten, entre muchas otras cosas, crear un apartado artístico impresionante de una forma relativamente sencilla.

Sin embargo, en comparación con otros motores que se verán más adelante, es bastante complicado de dominar debido a la complejidad de las herramientas que utiliza. Es por ello que para proyectos 2D o que sean pequeños es más recomendable si es posible decantarse por otros más sencillos y claros.

Uno de estos apartados es el lenguaje de programación, tratandose de C++. No obstante, para esto en concreto, se dispone de un sistema de programación visual denominado como *Blueprints* que facilitan la tarea a cambio de perder ciertas funcionalidades.

A continuación veremos tres de estas nuevas tecnologías integradas.

- Geometría virtualizada de Nanite[1] [2]: Básicamente es un sistema de renderizado que permite mantener en escena objetos con grandes cantidades de polígonos sin afectar al rendimiento. Tiene ciertas límitaciones, pero para la mayoría de casos proporciona una muy buena solución a este problema.
- Reflejos e iluminación global de Lumen[3]: Es el sistema de iluminación global y de reflejos que usa este motor, y que brinda una calidad altamente realista y preparada para las consolas de nueva generación.
- Mapas de sombras virtuales[4]: Permiten mantener sombras dinámicas de muy alta resolución incluso en zonas grandes y abiertas. Además, se puede complementar con los otros complementos vistos para funcionar a un rendimiento más que aceptable.

Evidentemente existen muchas más funcionalidades que se encuentran expuestas en la página oficial [2], pero las aquí tratadas sirven para mostrar el gran potencial de este motor.

[1] Página web con la documentación oficial sobre la geometría virtualizada de Nanite: https://docs.unrealengine.com/5.3/es-ES/nanite-virtualized-geometry-in-unreal-engine/

[2] Video de muestra de Nanite: https://www.youtube.com/watch?v=-50MJf7hyOw

[3] Página web con la documentación oficial sobre la Lumen: https://docs.unrealengine.com/5.3/es-ES/lumen-global-illumination-and-reflections-in-unreal-engine/

[4] Página web con la documentación oficial sobre los mapas de sombras virtuales: https://docs.unrealengine.com/5.3/es-ES/virtual-shadow-maps-in-unreal-engine/

2.1.2. Unity 3D

Figura 2.2: Logo de Unity

Unity es un motor desarrollado por Unity Technologies que ofrece muy buenas bases para el desarrollo tanto de juegos 3D como 2D. A diferencia de Unreal Engine, es más intuitivo y sencillo de entender, permitiendo a usarios con menos experiencia empezar más rápidamente.

En este caso, el lenguaje de programación que se usa para realizar los scripts es C#, siendo también bastante más cómodo y simple que C++. Sin embargo, aunque también ofrece muy buenas herramientas, muchas veces es necesario utilizar de terceros para ciertas tareas, lo cual puede llevar a problemas de incompatibilidad debido a actualizaciones. Además, en el apartado gráfico es inferior al Unreal, y para conseguir resultados similares se requiere de mucha dedicación.

Por otro lado, esta mejor preparado para realizar juegos para dispositivos Android y similares y es por todo esto que no se puede afirmar que sea peor ni mejor, simplemente existen diferencias a tener en cuenta para decidir por cual decantarse según el proyecto.

2.1.3. Game Maker Studio 2

Figura 2.3: Logo de Game Maker Studio

Una alternativa más sencilla a lo ya presentado podría ser Game Maker Studio 2, actualmente adquirido por YoYoGames. Se trata de un motor centrado en el desarrollo de videojuegos 2D que hace uso de un lenguaje de programación propio denominado Game Maker Language (GML). Sin embargo, de igual forma al Unreal, proporciona una alternativa de programación visual que, junto que diversas herramientas y métodos de trabajo basados en sistemas de arrastrar y soltar, facilitan a los usuarios sin experiencia programando la tarea de creación de videojuegos.

2.1.4. Otros

Hasta ahora se ha visto motores gratuitos que cualquier usuario puede descargar y usar. Dentro de esta categoría existen muchos otros que no se han nombrado como pueden ser Godot o Cryengine, pero también es interesante saber que en muchas ocasiones las empresas o grupos de trabajo pueden preferir crear y mantener sus propios motores.

Para esto último hay muchos ejemplos como:

- Rage Engine de Rockstar Games
- Id Tech de Id Software
- Creation Engine de Bethesda
- Source de Valve Corp.

En este trabajo realizado por el usario de GitHub *raysan5* [8] se pueden ver los motores que han utilizado muchas empresas y grupos de distintos tamaños en sus videojuegos, así como comentarios y apuntes del autor sobre el tema.

2.1.5. Herramienta elegida para el desarrollo del TFG

Para este proyecto se ha optado por utilizar Unity como motor para realizar el juego. Esta decisión se ha tomado por un lado en base a las necesidades técnicas, ya que se tratará de un videojuego con gráficos 2D y no demasiado complejo, por otro lado por la experiencia que ya poseo por haber trabajado con este motor en el desarrollo en otros trabajos, y finalmente porque existen herramientas como *ML-Agents* que permiten incorporar y utilizar redes neuronales en este entorno.

2.2 Herramientas de edición de imágenes digitales

Aunque para el objetivo principal del proyecto el apartado artístico no sea muy relevante, puesto que se va a realizar en base a un videojuego, es interesante darle cierta personalidad y carisma a través de este. Es por ello que es necesario hacer uso de herramientas de edición que permitan manipular y editar imágenes digitales.

2.2.1. Adobe Photoshop

Figura 2.4: Logo de Photoshop

Se trata de un software de pago desarrollado por Adobe Systems Incorporated, cuyo objetivo principal es la edición de fotografías. Cuenta con multitud de funcionalidades que permiten realizar tareas complejas de forma mucho más sencilla, y que además se pueden expandir con el uso de plug-ins. Finalmente, en lo que a aprendizaje se refiere, si que es cierto que para dominar este producto se requiere de mucha dedicación y horas, pero también contamos con una gran cantidad de tutoriales y documentación que pueden ayudar a entender y sacar el máximo provecho a este programa.

2.2.2. GIMP

Figura 2.5: Logo de GIMP

Por otro lado encontramos GIMP (GNU Image Manipulation Program) que tiene la misma finalidad que Photoshop, pero que a diferencia de este, es gratuito y de código abierto. Si los comparamos en los aspectos ya mencionados, también cuenta con buena cantidad de funcionalidades y plug-ins, aunque en menos cantidad y variedad, por lo que, pese a que se pueden obtener los mismos resultados o similares, el esfuerzo y conocimiento necesarios para obtenerlos será mayor. Por otro lado, existen una menor cantidad de tutoriales que puedan ayudarnos a aprender a utilizar este software, siendo necesario que el usuario no tenga más remedio que solucionar sus problemas por su cuenta, dificultando el aprendizaje.

2.2.3. Herramienta elegida para el desarrollo del TFG

En nuestro caso, debido a que únicamente será necesario este software para realizar pequeños y, en principio, sencillos retoques en ciertas imágenes, GIMP ofrece herramientas más que suficientes para llevar a cabo esta tarea.

2.3 Entornos de programación

En cuanto a entornos de programación, hay diversas alternativas que, teniendo en cuenta la decisión de trabajar en Unity y en el lenguaje de C#, se adecuan a esto y permiten progresar de una forma eficiente. Con esto en mente, se analizarán cuatro posibles opciones.

2.3.1. Visual Studio Code

Se trata de un editor de código fuente que soporta casi cualquier lenguaje de programación, cuyo funcionamiento se centra en la gestión de extensiones que amplían enormemente su funcionalidad. Es importante el hecho de que no se traté de una IDE, pues carece de una gestión de proyectos y otros elementos, de tal forma que ofrece un sistema más simple pero efectivo. Finalmente, en cuanto a la integración con Unity, existe un paquete que lo permite, no obstante actualmente ya no se mantiene y no esta actualizado.

2.3.2. Visual Studio 2022

En este caso sí que se trata de una IDE centrada en el desarrollo con lenguajes de C y similares. Siendo una opción más pesada, ofrece un buen sistema para gestión de proyectos software, contando con tres variantes:

- Comunidad: gratuita y pensada para estudiantes y desarrolladores independientes.
- Profesional: Una versión de pago con algo más de funcionalidad siendo más adecuada para equipos pequeños.
- Enterprise: Última versión de pago con la máxima funcionalidad ideada para ser escalable y utilizable por equipos de cualquier tamaño.

Para acabar, cuenta con una serie de extensiones que permiten una integración con Unity incluso a nivel de proyecto, tratándose de una muy buena opción que se mantiene actualmente.

2.3.3. JetBrains Rider

IDE diseñada para ser utilizada en el desarrollo de videojuegos, se trata de una herramienta perfectamente preparada para funcionar con Unity y con multitud de herramientas que facilitan y agilizan la producción de código. El único inconveniente notable es que no cuenta con versión gratuito, siendo un programa totalmente de pago.

2.3.4. Herramienta elegida para el desarrollo del TFG

Al final, en este proyecto se hará uso de Visual Studio 2022 debido a que su versión gratuita ofrece más que suficientes funcionalidades para gestionar las clases y el trabajo en general, siendo además fácil de integrar con Unity.

2.4 Inteligencia Artificial

Se trata de una campo muy complejo con multitud de conceptos e ideas que amplían enormemente la información según uno indaga más o menos en todo lo que se puede encontrar en la web o en libros de texto relacionados. Sin embargo, en este proyecto únicamente se tratará cual es el concepto base, un marco general sobre los últimos avances así como elementos de interés que existen actualmente, y su situación e importancia en el ámbito de los videojuegos.

Como concepto, la inteligencia artificial se puede interpretar como un algoritmo o conjunto de los mismos que tienen como objetivo el de imitar la inteligencia humana a la hora de realizar la tarea que se le encarga, evolucionando y mejorando a partir de la información que reciben.

2.4.1. Avances en la inteligencia artificial

En este apartado se verán dos campos en los que actualmente la inteligencia artificial a evolucionado muy rápidamente, produciendo además resultados impresionantes, y se acabará con el concepto de la inteligencia artificial general.

Modelos de generación de lenguaje natural

Desde hace unos años han ido ganando popularidad las redes que son capaces de procesar lenguaje natural y de dar una respuesta lógica, también en lenguaje natural, siendo los más conocidos *ChatGPT* o *Microsoft Bing*. En el caso de este primero, basándose en técnicas de refuerzo y supervisadas para entrenarse a partir de una gran cantidad de datos de texto, es una herramienta muy útil como asistente en multitud de tareas tales como escribir documentos, obtener información (aclarar que es importante valorar que puede cometer errores en sus respuestas), generar código...

Modelos texto a imagen

Otros modelos que también se están haciendo notar son aquellos capaces de generar imágenes a partir de texto, donde se pueden remarcar *Dall-E* y *Midjourney*. Este tipo de modelos también son capaces de procesar lenguaje natural pero, en lugar de dar respuestas, lo utilizan para generar imágenes de la nada según interpreten el mensaje recibido.

Inteligencia artificial general

Este último tipo de inteligencia artificial no es algo que este ni mucho menos cerca de estar desarrollado, tratándose por definición de lo que representaría una inteligencia artificial perfecta capaz de, incluso, igualar o superar al humano en cualquier tarea que requiera de intelecto que se le proponga, a diferencia de las vistas hasta ahora que están centradas alrededor de un objetivo. La IAG plantea muchos dilemas éticos, además de ser considerado un posible factor de riesgo para los propios humanos

2.4.2. Inteligencia artificial en los videojuegos

Para este trabajo, en el campo de los videojuegos se distinguirán entre dos formas de inteligencia artificial: *Behaviour Selection Algorithm* y *Machine Learning*.

Behaviour Selection Algorithm

Se trata de la más común actualmente en el desarrollo de videojuegos, basándose en la selección de unos comportamientos u otros en función de un algoritmo. Algunos de los más importantes son:

- Máquinas de estado finito
- Árboles de decisión
- Árboles de comportamiento

Machine Learning

Esta otra forma de inteligencia artificial se basa en la producción y entrenamiento de modelos a partir de datos e información. Aunque no se suele utilizar en la mayoría de casos por su complejidad, existen ciertos campos como el ajedrez o juegos complejos de estrategia donde si se trata de una opción a tener en cuenta.

CAPÍTULO 3
Marco teórico

En este apartado se presentarán algunos conceptos de elementos que aparecerán a lo largo del proyecto y que no se explican en detalle. Así, con esto se pretende que se entienda lo mejor posible el contenido del trabajo.

3.1 Aprendizaje por refuerzo

Dentro de las distintas formas de aprendizaje y debido a la naturaleza del problema, siendo imposible obtener datos etiquetados que se puedan utilizar para entrenar y, además, teniendo en cuanta que se busca un objetivo específico (la victoria del agente en la partida), se utilizará el aprendizaje por refuerzo. Existen otras formas de atajar el problema, como los algoritmos evolutivos, sin embargo a causa del tipo de videojuego realizado y de como se preparó para esta parte, no serían una buena solución.

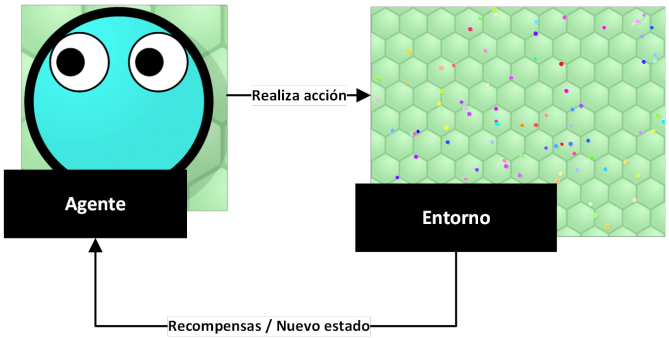


Figura 3.1: Representación del aprendizaje por refuerzo con ilustraciones del videojuego.

Con esto decidido, el aprendizaje por refuerzo sigue un esquema al de la figura 3.1, donde el agente a partir del estado del entorno y una serie de recompensas, decide que acción realizar que cambie el actual estado. Con este ciclo, el agente va evolucionando buscando siempre la acción que le genere la mejor recompensa.

3.2 Funcionamiento y conceptos de elementos de Unity

Para acabar con estos conceptos introductorios, se verá una pequeña explicación de las bases del funcionamiento de Unity, así como un par de conceptos interesantes para entender ciertas partes de la documentación.

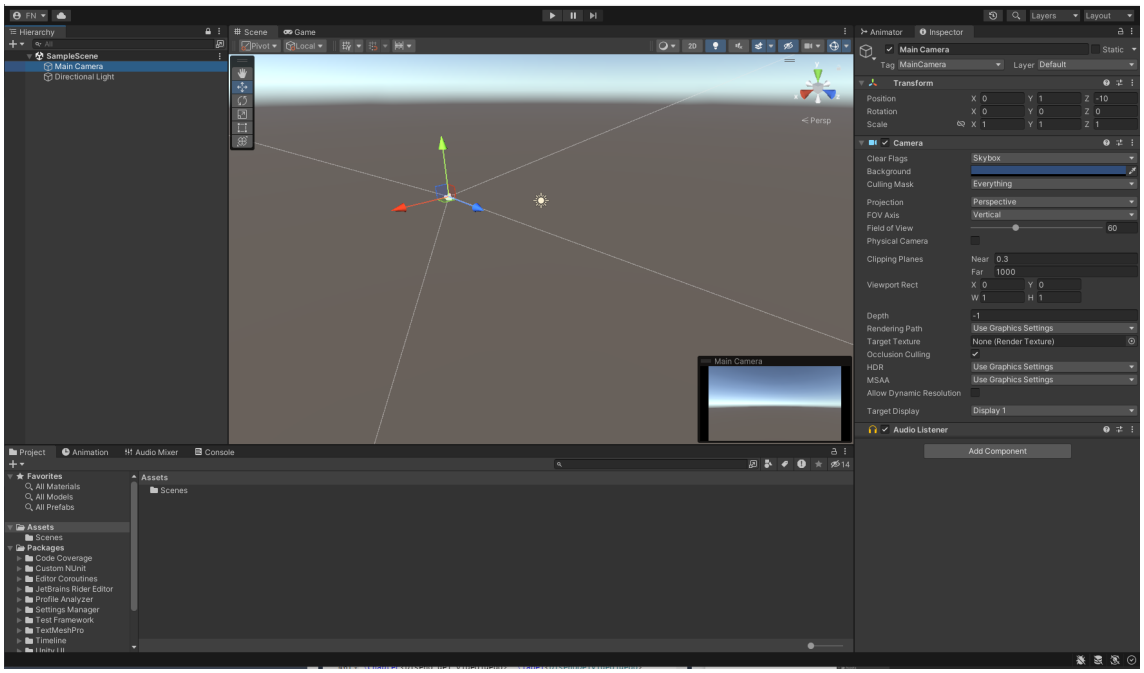


Figura 3.2: Escena vacía en Unity.

De la parte de trabajar en Unity es importante conocer que se basa en manejar una jerarquía de objetos, compuestos a su vez por componentes. Así, en la figura 3.2 se puede ver una escena vacía al iniciar Unity, donde se pueden diferenciar 4 grandes ventanas principales. A la izquierda se encuentra la jerarquía ya nombrada, donde se pueden ver las relaciones padre-hijo de todos los elementos que conformaran la escena. En medio se encuentra una visualización 3D o 2D, depende de como se configure, del estado actual del videojuego. A la derecha se encuentra el inspector donde se pueden ver todos los componentes de un objeto seleccionado. Finalmente, abajo se encuentra el sistema de ficheros del proyecto donde se almacenan todo aquello que se vaya a usar.

En cuanto a los conceptos, se verán dos:

- *Prefab*: Se puede ver como una plantilla que contiene todos los componentes e información de un objeto en Unity. Con este tipo de elementos se puede instanciar múltiples copias de un mismo objeto, todas independientes pero enlazadas al propio *Prefab*, de tal forma que si este sufre cambios, todas las copias también.
- *Scriptable Object*: Es un tipo de elemento útil para almacenar información única. Así, cuando se instancia un *Scriptable Object*, su contenido se mantendrá siempre en esa sesión de Unity, incluso entre escenas.

CAPÍTULO 4
Diseño del videojuego

Puesto que en el apéndice D se encuentra toda la información necesaria con los conceptos y elementos principales que conforman el videojuego, aquí se tratarán los aspectos más técnicos relacionados con la programación y desarrollo en Unity. Para ello, dividiremos la información en dos secciones, una para cada una de las escenas principales: menú principal y en partida. Además, como todo el código se encuentra visible y comentado mayormente en los enlaces al *git* en el apéndice B, aquí se tratará también de una forma más global, mostrando como todo interactúa entre sí, aunque para todavía más información sobre esto también se puede ver el apéndice A.

4.1 Escena del menú principal

Se trata de la primera pantalla con la que interactúa el jugador, y a partir de la cual podrá decidir entre los distintos modos de juego, ajustar diferentes opciones, consultar los controles y los créditos, o cerrar la aplicación.

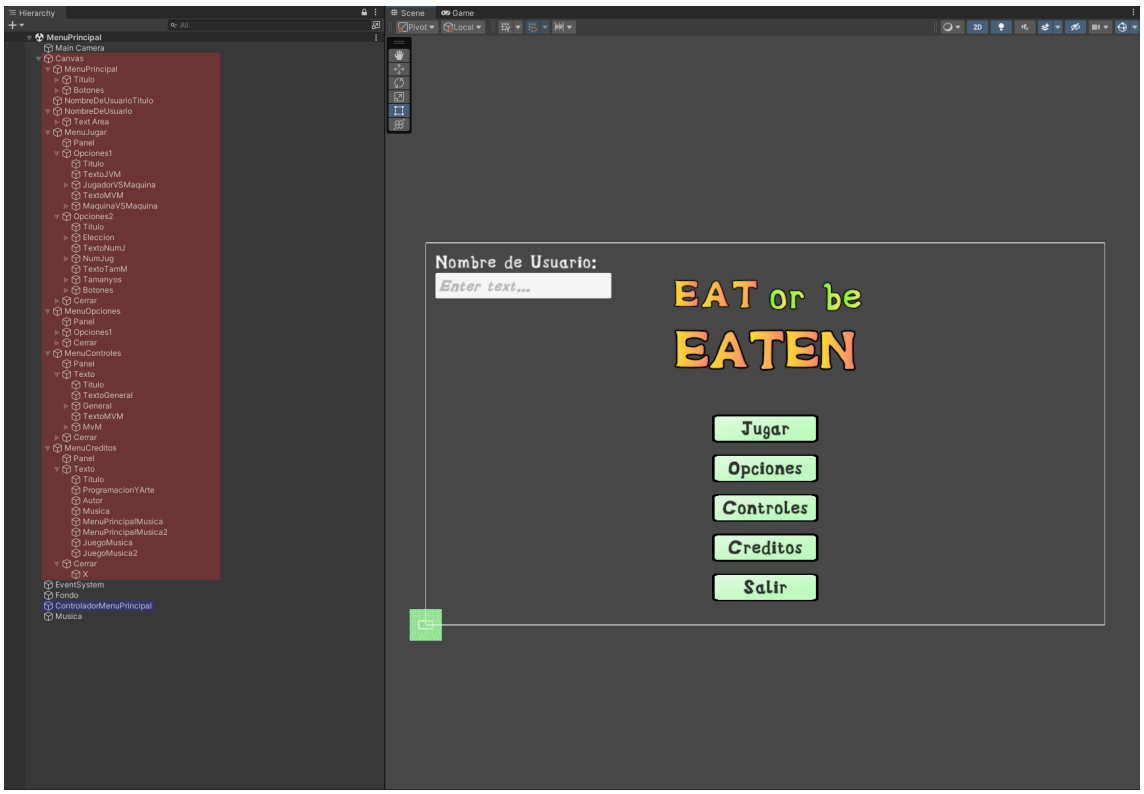


Figura 4.1: Menú principal desde el desarrollador de Unity

En términos de programación, solo cabe destacar el código del controlador de menú principal 5*b*, que se encarga de mantener todos lo métodos que usarán los distintos elementos de la interfaz y que se encuentra en el componente remarcado de azul en la imagen 4.1, así como la estructura necesaria para mantener el orden en la interfaz como se puede observar en el remarcado rojo.

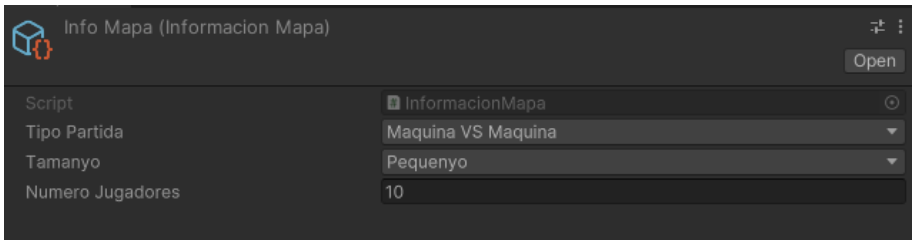


Figura 4.2: *Scriptable Object* con la información de la partida

Además, para transmitir la información de la escena del menú principal a la de juego, se hace uso de un *Scriptable Object* como se muestra en 4.2, que se modifica según lo indicado por el usuario.

4.2 Esquema del menú principal

En la figura 4.3 se puede ver una versión simplificada descartando los componentes menos importantes de la jerarquía que sigue el menú principal, donde los bloques negros representas componentes de Unity y los que tiene el texto verde elementos de *scripts*:

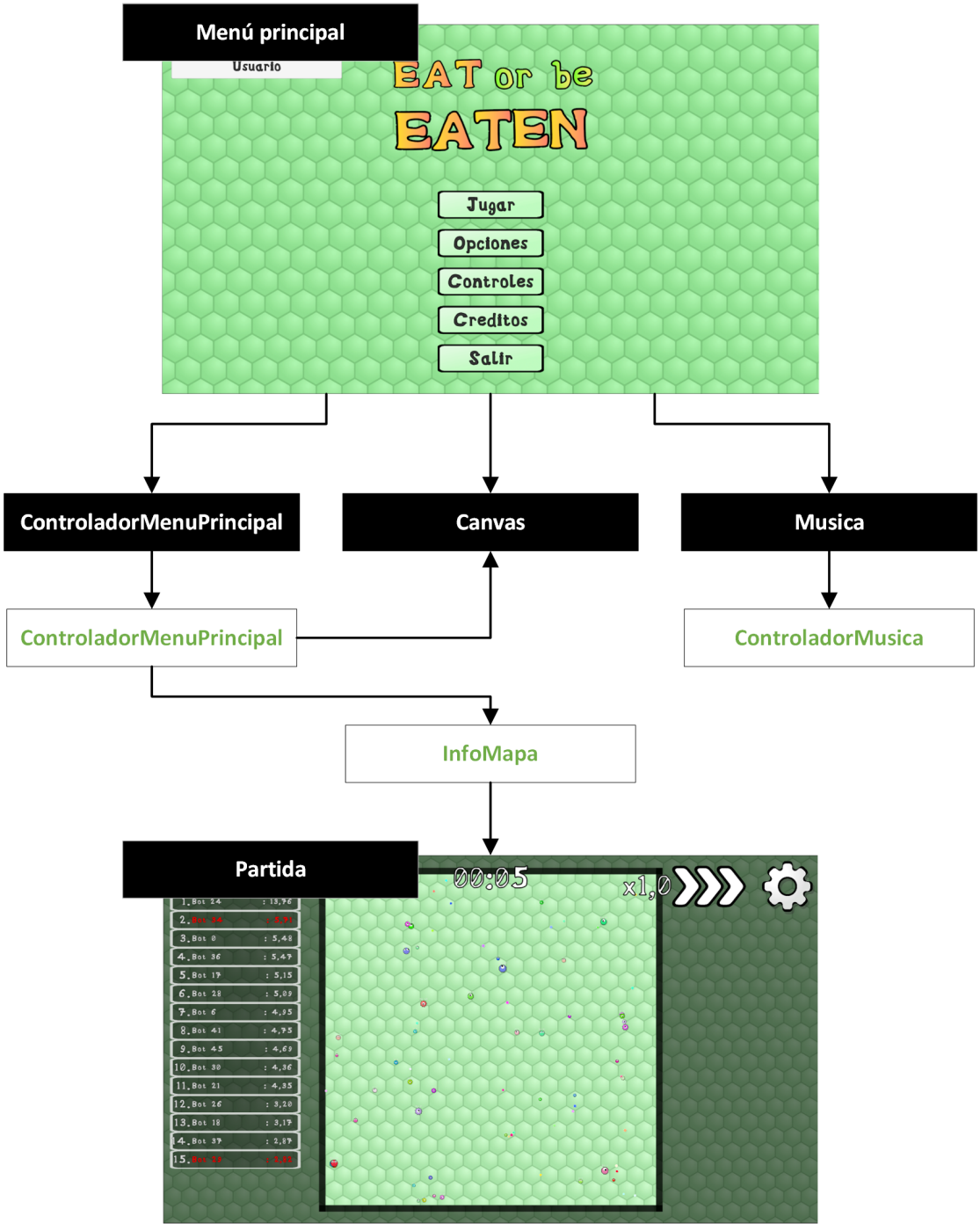


Figura 4.3: Esquema básico del menú principal.

Con este esquema puede verse de mejor forma la idea detrás del menú principal, sirviendo como un punto previo a la partida donde se deciden los parámetros que la dirigirán.

4.3 Escena de una partida

Esta es aquella en la que se desenvuelve el *gameplay*, siendo bastante más compleja que la anterior descrita.

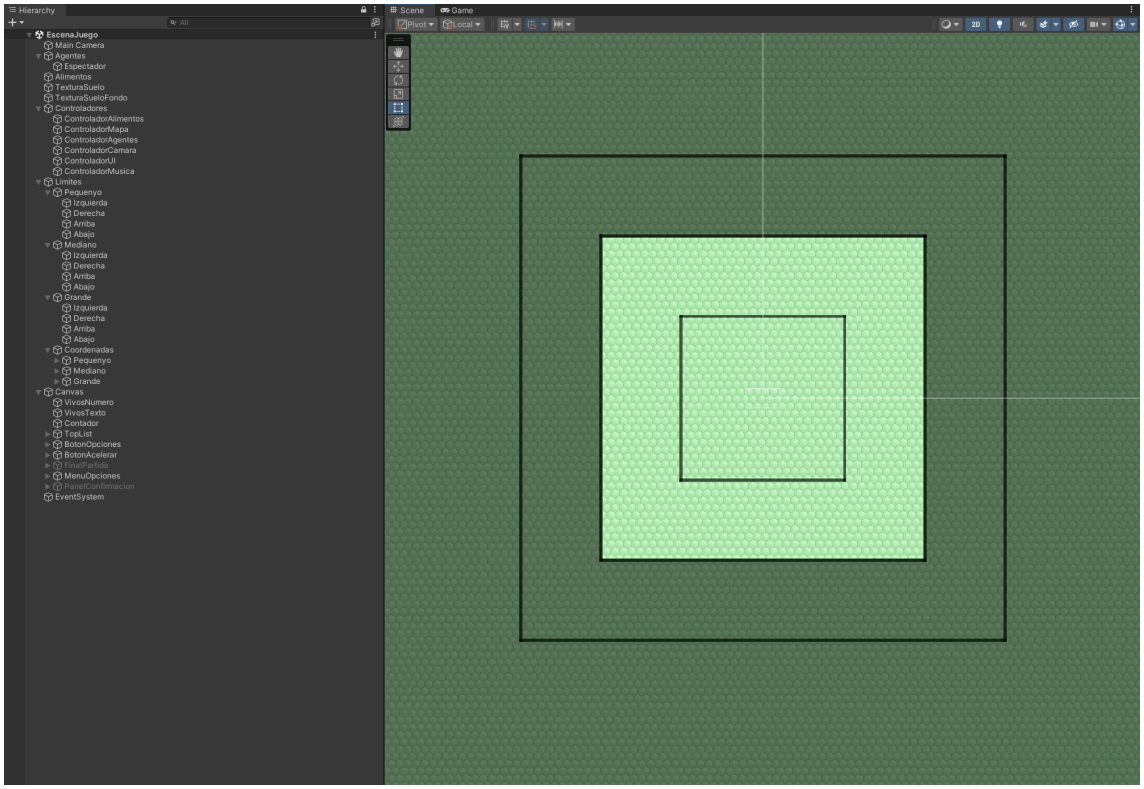


Figura 4.4: Escena donde se desarrolla las partidas en Unity

En la imagen 4.4 se puede observar nuevamente la jerarquía que organiza dicha escena, donde caben destacar los componentes de:

- Agentes: Sirve de contenedor donde se crearán y mantendrán todos los agentes necesarios e indicados por el usuario previamente. Además, contiene el elemento espectador utilizado en caso de que el tipo de partida sea de máquina contra máquina. En el apartado 4.3.1 se tratará más en detalle como funcionan los agentes.
- Alimentos: De igual forma, sirve como contenedor para almacenar y manejar todos los alimentos que puedan ser necesarios en una partida. Siguiendo la técnica de *object pooling*[1], se crean todos desde el principio de la partida y según se vayan requiriendo se colocan y muestran, mientras que cuando son destruidos por una agente se ocultan y quedan a la espera de ser utilizados nuevamente en un futuro. Gracias a esto se optimiza el uso de la CPU reduciendo la cantidad de creaciones y destrucciones de objetos que se realizan. En el apartado 4.3.3 se explicará el funcionamiento concreto de los alimentos.
- Controladores: En este elemento se encuentran a su vez los componentes que contendrán los *scripts* que controlen componentes clave de la partida. La explicación de cada uno de estos componentes se encontrará en el apartado 4.3.4.

[1]Más información sobre *object pooling* en: https://en.wikipedia.org/wiki/Object_pool_pattern

- Límites: Contiene los bordes negros visibles en la escena que, además de servir como elemento visual y práctico (bloquea a los agentes) durante el *gameplay*, también proporciona información útil a los *scripts* a la hora de, por ejemplo, colocar elementos en el mapa.
- *Canvas*: Finalmente se encuentra el *canvas*, que se trata de la jerarquía con los elementos que componene la interfaz.

En la figura 4.5 se puede ver una representación en forma de esquema de lo explicado en los puntos anteriores. Igual que con el esquema del menú principal, los elementos negros representan objetos de Unity, mientras que los de texto verde son *scripts*. Otra cosa a tener en cuenta es que se obvian elementos de poco interés, así como ciertos enlaces innecesarios, ya que no se pretende tener una representación totalmente correcta y perfecta, sino una que muestre de forma visual lo ya explicado.

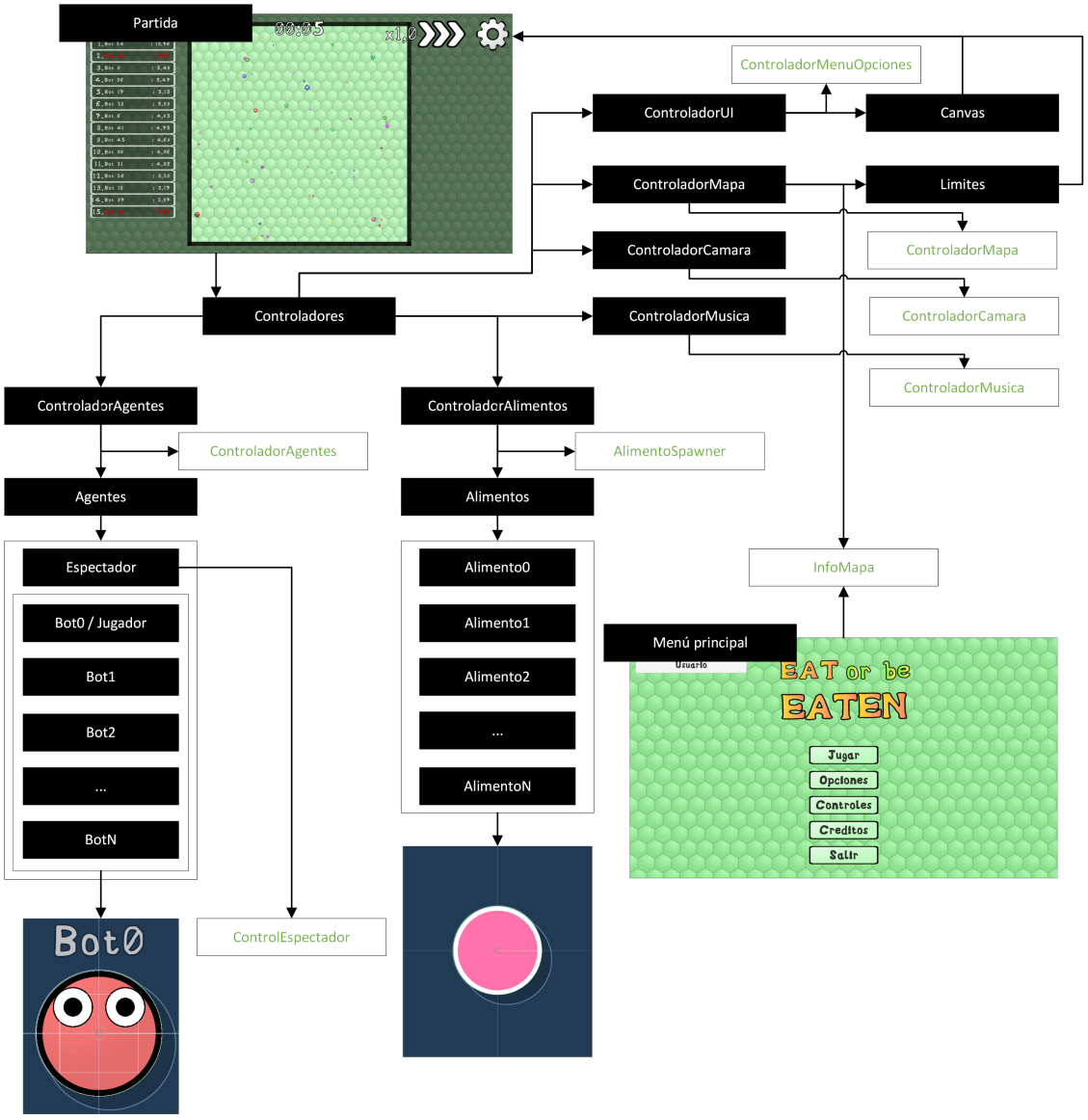


Figura 4.5: Esquema simplificado de la escena de una partida.

Para acabar con la presentación del juego inicial, se tratarán los tres elementos clave que faltan:

- Funcionamiento de los agentes
- Funcionamiento de los alimentos
- Funcionamiento del mapa

4.3.1. Funcionamiento de los agentes

Los agentes son los principales y únicos personajes del videojuego, siendo a su vez el elemento clave que dirige el progreso de las partidas. Es por ello que se trata de un componente con multitud de funcionalidades y acciones relativamente complejas.

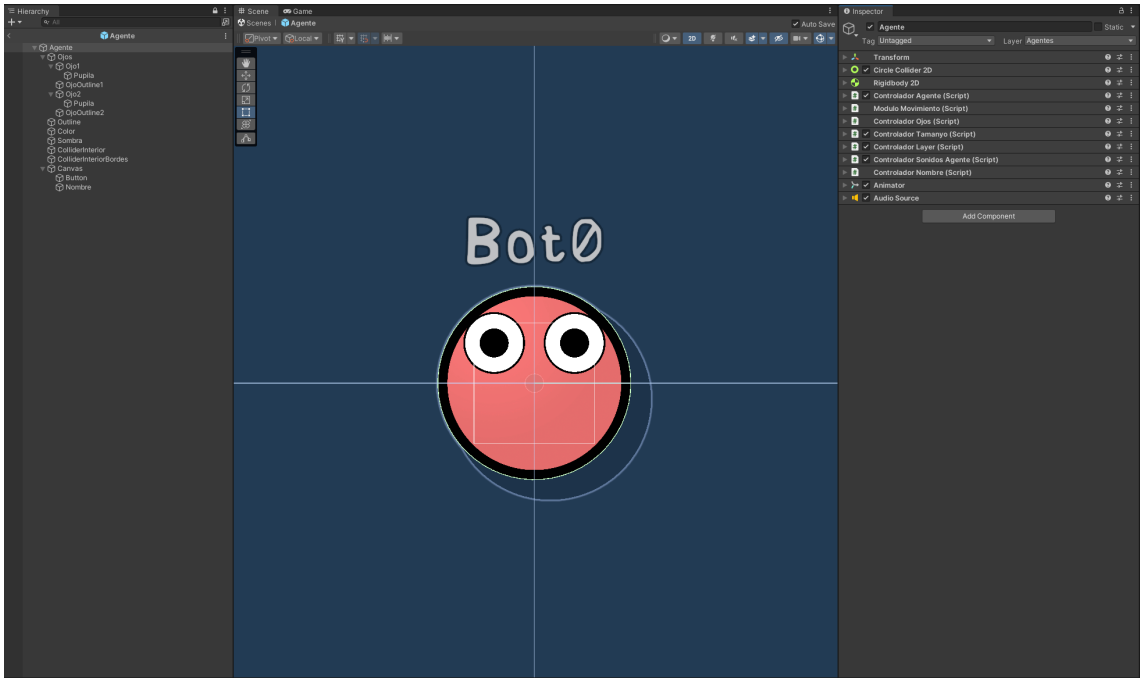


Figura 4.6: *Prefab* del agente en Unity

En la imagen 4.6 se puede ver al agente en Unity, tratándose de un *prefab* padre del cual existen dos hijos para cada tipo de agente extra (en total hay 3), de tal forma que la única diferencia entre ellos es visual, manteniendo la misma funcionalidad. Además, para facilitar su desarrollo, se ha programado teniendo en cuenta una división en lo que se podrían catalogar como módulos, de tal forma que cada uno de estos llevan a cabo una función o acción específica, teniendo la capacidad de comunicarse entre ellos. En la imagen se encuentran como *scripts* en la sección derecha del inspector, donde se puede diferenciar un total de 7 módulos distintos.

ControladorAgente

Entre otras tareas auxiliares y específicas, primordialmente, este código se encarga de centralizar todos los módulos principales de un agente, gestionando sus funcionalidades. Más concretamente, gestiona el movimiento del agente, su estado actual en la partida (si ha sido eliminado o no, su tamaño...) y calcula los *ray casts* necesarios para la IA.

ModuloMovimiento

Este módulo devuelve un vector de dirección cuando se le pida a partir de los resultados de los *ray casts* y según el tipo de agente que maneja. Así, si se trata del jugador realiza un simple cálculo en función de los inputs del usuario, y si se trata de la IA programada hace uso de la clase 1*h* para calcularla. En la versión inicial, pese a no incluir la IA que hace uso de la red neuronal, esta planteado con la idea de añadir una nueva entrada que permita esta opción.

ControladorOjos

Como se especifica en el GDD (D), para crear una mejor inmersión y darle algo de vida a los agentes, estos mueven los ojos en función de su movimiento. Este módulo se encarga de controlar esta funcionalidad, de tal forma que el *ControladorAgente* solo debe indicarle la dirección de movimiento actual para que actualice el estado de los ojos.

ControladorTamanyo

El tamaño de los agentes es una de las mecánicas principales de este videojuego, pues es lo que decide que agentes eliminan a que agentes. En la práctica se maneja un valor llamado *Volumen* que indica como de 'grande' es un agente y si uno lo es más teniendo en cuenta un pequeño margen porcentual, entonces puede eliminar al otro. Visualmente, para controlar cuanto ocupan los entes en pantalla, en vez de ser una relación lineal entre *Volumen* y tamaño físico, se ha optado por aplicar primero una raíz cuadrada, de tal forma que los números son mucho menores pero se puede apreciar la diferencia en el *gameplay*. De forma similar al *ControladorOjos*, este controlador recibe en todo momento indicaciones del *ControladorAgente* para actualizar el valor de *Volumen* y mantener la concordancia visual.

ControladorLayer

Para resolver el problema de que agentes se dibujan sobre que agentes y de paso ayudar a distinguir mejor cuales son más grandes que otros, se ha tomado la decisión de que los de mayor tamaño siempre se dibujen por encima de los más pequeños. Este módulo se encarga de mantener esto haciendo uso del sistema de capas que ofrece *Unity* a la hora de renderizar elementos 2D. Puesto que dicho sistema es algo limitado y solo permite diferenciar entre 32767 (máximo valor de un *short*) posiciones en una misma capa, ha sido necesario hacer uso de múltiples capas entre las que van cambiando los agentes. Así, se ha conseguido que el nivel de precisión con el que se puede diferenciar un agente de otro pueda reconocer hasta dos decimales del valor de su *Volumen*. A diferencia de los módulos anteriores, este se comunica directamente con *ControladorTamanyo* para obtener el valor de *Volumen* directamente y actualizar lo necesario en su propio *Update*.

ControladorSonidosAgente

Esta clase es la encargada de reproducir cualquier sonido que deba hacer el agente. En este caso es tan simple como un '*pop*' aleatorio, que sonará cuando haya sido eliminado siguiendo las ordenes de *ControladorAgente*.

ControladorNombre

Por último se encuentra el módulo encargado de controlar la visibilidad y contenido de las etiquetas de nombre que poseen los agentes. El contenido se establece al inicio de la partida (todos los bots y el jugador si es necesario), y a lo largo de la partida puede cambiar si el agente es eliminado (debe desaparecer) o si el usuario lo indica pulsando la tecla Q. Es por esto último que también *ControladorAgente* maneja las condiciones e indica que hacer a *ControladorNombre*.

Además de estos módulos, también podríamos relacionar con los agentes los controles del jugador que incluyen el código del control de la cámara 4*a* y control del espectador 6*a*, así como la IA programada 1*h* y el objeto *ray cast* 1*j*. Se tratará ahora todos a excepción de la IA programada, ya que se explicará detalladamente en el apartdo 5.1.

ControlCamara

La cámara esta programada de tal forma que siempre debe seguir a un ente, ya sea un agente en el caso de que el jugador controle o este espectando a uno, o al agente espectador, que se trata de uno no visible cuya única funcionalidad es la de servir como referente a la cámara. Por otro lado, también controla la posibilidad de realizar zoom en caso de seguir al ente espectador y se asegura de que este permanezca donde se encuentra el posible agente que siga la cámara para que, si se decide dejar de seguirlo, la cámara prosiga de forma natural desde donde se abandona al agente.

ControlEspectador

Este código se encarga del comportamiento del ente espectador presentado en el apartado anterior. Concretamente, del funcionamiento del zoom y de su movimiento cuando se especta.

ObjetoDeRayCast

Finalmente, *ObjetoDeRayCast* se usa para encapsular toda la información que puede ser interesante y que pueda aportar un objeto detectado por un *ray cast*. En concreto, almacena cuatro parámetros:

- Tipo de objeto, que puede ser: un agente, un alimento, un borde o nada.
- Posición donde se encuentra el objeto.
- Distancia a la que se encuentra el objeto.
- Tamaño del objeto si procede.

4.3.2. Esquema del agente

En la figura 4.7 nuevamente se encuentra un esquema, esta vez del agente, donde lo más importante son los enlaces entre los propios *scripts*, puesto que representan la idea de como deberían funcionar estos módulos, mientras que los enlaces del agente a los *scripts* representan que estos últimos son componentes explícitos del primero.

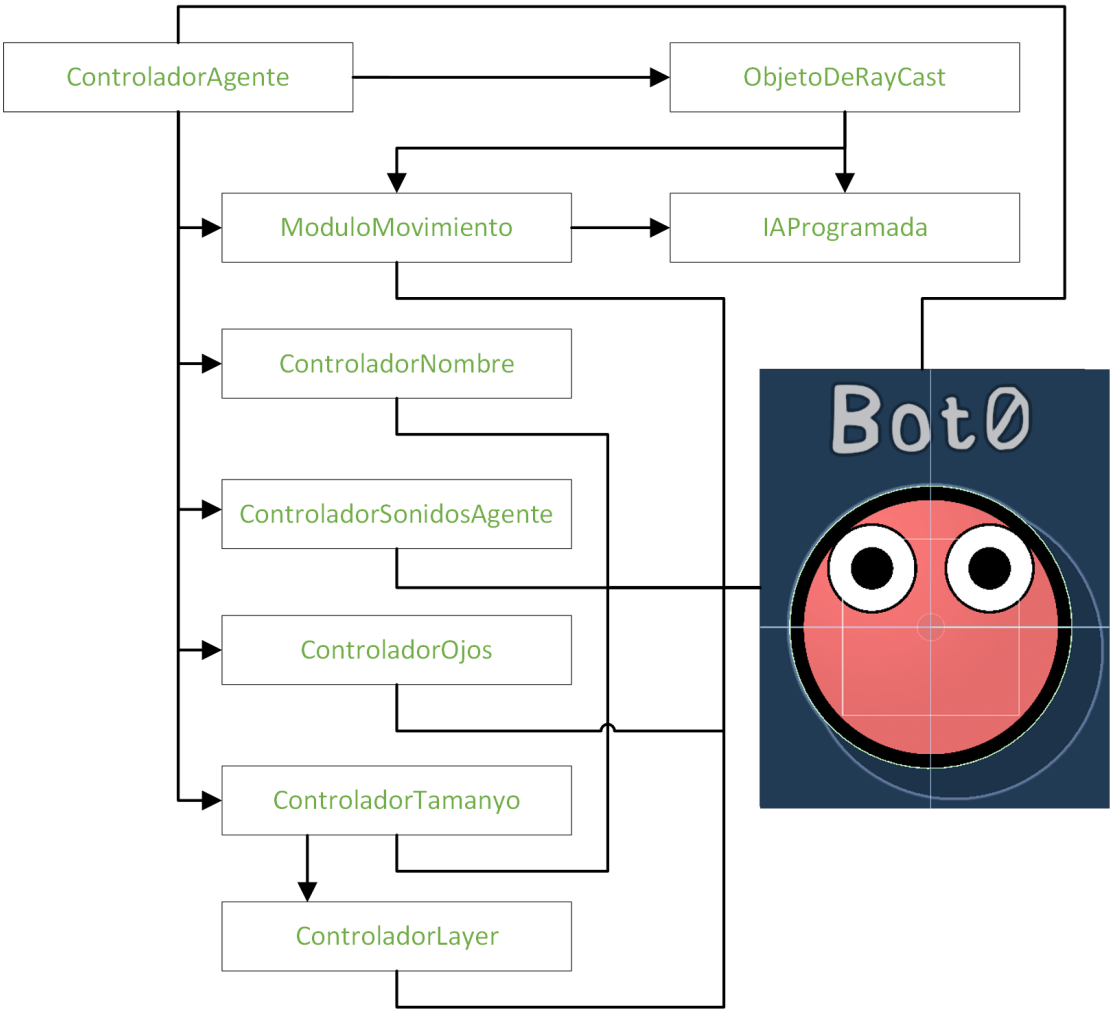


Figura 4.7: Esquema simplificado de un agente.

4.3.3. Funcionamiento de los alimentos

Otro componente primordial para el correcto desarrollo del juego son los alimentos estáticos que van apareciendo aleatoriamente en el mapa, ya que sirven como sustento principal para que los agentes vayan aumentado de tamaño, avanzando así la partida.

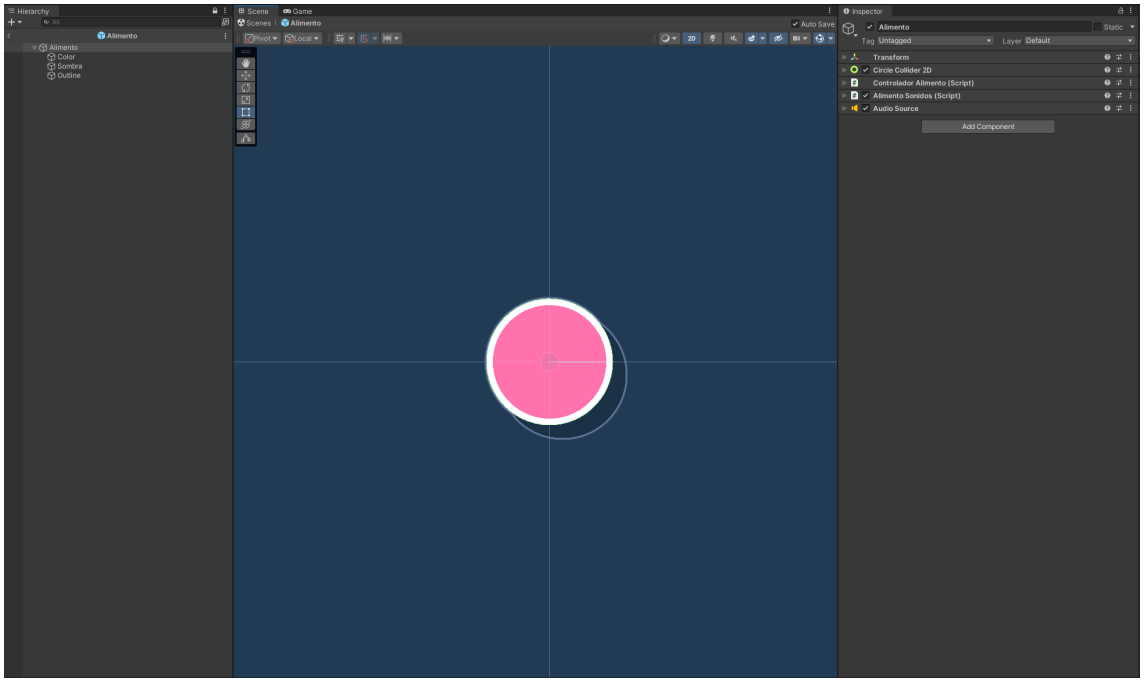


Figura 4.8: *Prefab* del alimento en Unity

Nuevamente se trata de un *prefab* como se puede ver en la imagen 4.8, y en este caso contamos con dos módulos que ejecutan el comportamiento de estos.

AlimentoSonidos

De forma equivalente al agente, esta clase se encarga de reproducir los sonidos que pueda producir un alimento. Igualmente se trata de un '*pop*' aleatorio cuando entra en contacto con un agente y es destruido.

ControlAlimento

Este código se encarga principalmente de controlar el comportamiento de un alimento cuando interactúa con un agente. Así, informa al agente de que lo ha consumido en caso de ser posible y procede a auto destruirse siguiendo una animación llevada a través del código que simula al agente absorbiéndolo.

4.3.4. Funcionamiento del mapa

De forma similar a con los agentes, el mapa y el desarrollo de la partida son manejados por diversos controladores con funcionalidades determinadas, que también se comunican entre sí e incluso con código ya visto en apartados anteriores. Así, contamos con los siguientes controladores:

ControladorAgentes

En el desarrollo del juego, *ControladorAgentes* se encarga de organizar a todos los agentes, así como de eventos importantes relacionados con estos, gracias a que cuenta con un *array* con todos los que se encuentran en partida. Concretamente realiza las siguientes acciones:

- Mantiene actualizada la lista de la interfaz con los mejores agentes.
- Prepara a todos los agentes en función de los parámetros establecidos por el usuario al comenzar la partida.
- Se encarga del proceso de eliminar agentes, incluyendo la animación de 'absorción'.
- Ajusta las velocidades de los agentes, de tal forma que el agente más pequeño siempre vaya a la velocidad inicial, y los más grandes se calcule en función de esta última.
- Ayuda al *ControladorCamara* con el proceso de selección de agentes mediante el ratón cuando se especta.

AlimentoSpawner

Esta clase principalmente se ocupa de manejar todos los alimentos de la partida, siguiendo el ya explicado método de *object pooling*. Los crea al inicio de la partida, y los coloca periódicamente y aleatoriamente en el mapa. Dicho periodo se reduce según avanza la partida en función del tiempo, de tal forma que se consigue que cuanto más grandes sean los agentes, más alimentos haya en el campo.

ControladorMapa

La tarea de *ControladorMapa* es la de preparar el entorno donde se va a desarrollar la partida. En concreto, activa los bordes necesarios que se pueden ver en 4.4 e informa al resto de *scripts* sobre estos límites. Toda la información la obtiene del ya nombrado *scriptable object* (imagen 4.2). Además, mantiene actualiza el contador que se encuentra en la interfaz.

Para acabar, en este apartado también se puede tratar todo el código involucrado en el manejo de interfaces durante una partida.

SonidoBotones

Se encarga de proporcionar un método al que pueden acceder los botones para reproducir un sonido cuando son presionados.

ControlMenuOpciones

Contiene los métodos para todas las acciones que puedan llevar a cabo los elementos de la interfaz, y además maneja los inputs de teclado del usuario para abrir el menú de opciones y ocultar la lista de mejores agentes.

MenuSonidos

De igual forma a *SonidoBotones*, proporciona un método para reproducir un sonido cuando los distintos paneles del menú se ocultan o muestran.

ControladorMusica

Se encarga de reproducir aleatoriamente una de las músicas proporcionadas. Además, se asegura de que al acabar comience una nueva, consiguiendo que siempre se este escuchando alguna.

CAPÍTULO 5
Implementación de las redes neuronales

En este capítulo se tratará todo el proceso realizado para incorporar redes neuronales que doten de inteligencia artificial a nuestros agentes. Aunque, primeramente se verá como esta desarrollada la IA actual que guía a los agentes en el estado inicial del videojuego.

5.1 Funcionamiento de la IA programada

La versión inicial del juego cuenta con una inteligencia artificial que, en función de los objetos detectados a través de los *Raycasts*, decide en que dirección avanzar. En el apéndice D del GDD existe un apartado que trata de explicar su funcionamiento haciendo uso de una máquina de estados. No obstante, como también se menciona en dicho apartado, esto es solo una representación, ya que en realidad el código simplemente toma una decisión a partir de la información actual sin tener en cuenta ningún estado. Pese a tener acceso al código, quizá sea interesante explicar como funciona.

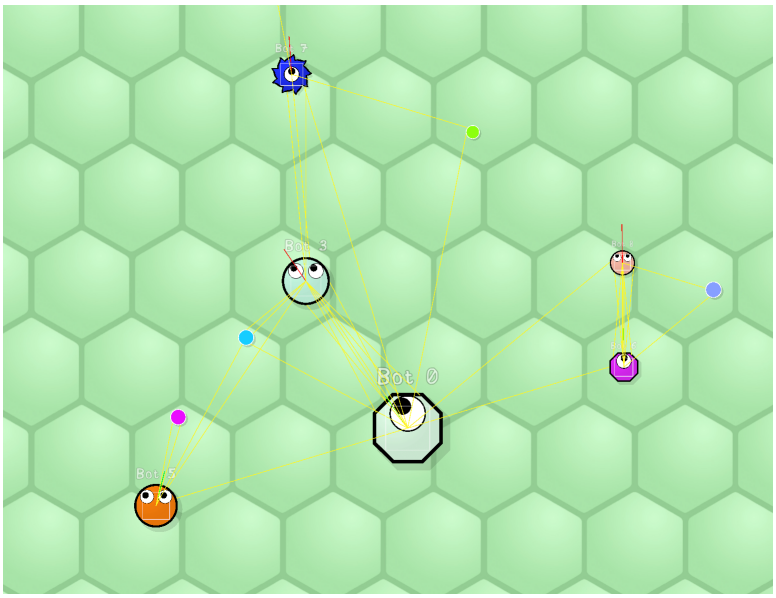


Figura 5.1: Diversos agentes en partida

En la figura 5.1 se puede observar una partida con el *debugging* activo donde se muestra diversos agentes y sus respectivos *raycast* que han alcanzado objetivos, además de un

vector verde o rojo que indica la dirección a la que se dirige el agente. El color de dicho vector se utiliza como indicativo de la intención del agente:

- **Verde**: Buscando un alimento o un agente de inferior tamaño.
- **Rojo**: Huyendo de un agente peligroso para el agente.
- **Azul**: Movimientos aleatorios debido a la ausencia de información.

Así, en el código de la IA, se realiza primeramente un análisis de los objetos detectados con los *raycasts*, guardando información de lo que sería el mejor alimento (teniendo en cuenta que agentes inferiores son 'alimentos') y el peor agente detectados basándose en el tamaño y la distancia, ambos ponderados. Tras esto, en función de si existen uno, los dos o ninguno, se realiza lo siguiente:

- **Ninguno**: Se calcula un vector aleatorio que se actualiza periódicamente. Además, en caso de detectar paredes a partir de una distancia máxima y que se encuentran en la dirección de movimiento, el vector se recalcula evadiendo la pared en una dirección aleatoria.
- **Solo alimento**: Se calcula el vector que va del agente al mejor alimento, avanzando así en línea recta en su dirección.
- **Solo agente**: Se calcula el vector opuesto del que va del propio agente al agente enemigo. Además, si se encuentran paredes en la dirección de movimiento, se tratarán de evitar, huyendo en una dirección que no nos acerque demasiado al enemigo según el ángulo. Debido a esto, en esta versión cuando se encuentra contra una esquina, el agente se bloquea debido a que ninguna posibilidad cumple esta última condición.
- **Ambos**: Igual que el anterior, pero se valorará la posibilidad de desviarse a por el alimento durante la huida.

La idea de haber realizado el código en base a los *raycasts* pese a posiblemente no ser tan óptimo y, en algunos casos, complicar ciertas situaciones, es debido a que se consigue demostrar que tan solo con esa información el agente debe ser capaz de tomar una decisión que le permita ser competente en el juego. De hecho, debido a lo rápido que deciden, para una persona es en ciertas situaciones difícil de esquivarlos, aunque con el conocimiento adecuado se les puede hacer frente sin problemas.

5.2 Análisis de las alternativas

Tras una búsqueda en la web, se puede encontrar *ML-Agents* a la cabeza en lo que a redes neuronales en Unity se refiere, además de otras alternativas no tan conocidas o realizadas por usuarios independientes. Entre ellas encontramos las siguientes:

- Librería de *Gym*[1]: Una librería de Python específicamente diseñada para crear comunicaciones entre agentes y entornos. Pese a ser buena opción en cuanto a que sirve como base para realizar pruebas y estudios, se trata de algo de más bajo nivel que implicaría realizar tareas extra fuera de lo planteado.
- Red neuronal dentro de Unity desarrollada por *Blueteak*[2]: Un proyecto elaborado en Unity que implementa una red neuronal simple. En este caso no se adaptaría correctamente al videojuego, puesto que según presenta los ejemplos esta preparada para entornos muchos más simples en comparación al de este trabajo.
- Opciones con algoritmos genéticos[3] [4]: Por último, existen opciones que hacen uso de algoritmos genéticos para entrenar a las redes y, pese a que podría ser una opción interesante a tener en cuenta, la naturaleza del videojuego complica demasiado su utilización.

Al final, todas estas últimas no suelen ofrecer sistemas que permitan a las redes aprender a través de técnicas por refuerzo o requieren de un esfuerzo que se considera fuera del contenido de este proyecto. Es por ello que, tras indagar en el funcionamiento más especifico de *ML-Agents*, se puede ver que ofrece multitud de herramientas que permiten que el entrenamiento de la red sea más diverso, pudiendo aportar resultados y conclusiones bastante interesantes.

Así, después de leer más detalladamente la documentación oficial que se encuentra en [10] y valorar las diferentes opciones en función de la tarea que se desea realizar, podemos encontrar tres alternativas:

- Aprendizaje por refuerzo
- Aprendizaje por imitación mediante *Generative Adversarial Imitation Learning* (GAIL)
- Aprendizaje incorporando memoria con *Stacked vectors* o *Long Short-Term Memory* (LSTM)

Para comprobar que efectivamente este tipo de aprendizajes se usan en este ámbito, se ha realizado una pequeña búsqueda de artículos que realicen algún tipo de estudio del tema. Concretamente para refuerzo es fácil encontrarlos, siendo [11], [15] y [13] algunos ejemplos (algunos indican el uso de *Deep Reinforcement Learning* que, pese a ser claramente distinto puesto que se usa en entornos más complejos, las bases son similares al entrenamiento por refuerzo). En cuanto al uso de GAIL también encontramos algunos como [12] y [14]. No obstante, para LSTM no se ha podido encontrar documentación que pruebe este uso, así que es necesario conformarse con la documentación oficial en [5].

[1] Enlace a la API de *Gym*: https://www.gymlibrary.dev
[2] Proyecto de *Blueteak* con el proyecto: https://github.com/Blueteak/Unity-Neural-Network
[3] Proyecto de *EricBatlle*: https://github.com/EricBatlle/UnityNeuralNetwork
[4] Proyecto de *HectorPulido*: https://github.com/HectorPulido/Evolutionary-Neural-Networks-on-unity-for-bots

A continuación, para cada tipo de entrenamiento se dará una breve explicación de lo que se pretende hacer y obtener en este proyecto:

5.2.1. Aprendizaje por refuerzo con ML-Agents

El primer entrenamiento únicamente se llevará a cabo haciendo uso de la técnica de aprendizaje por refuerzo. Con estas pruebas se obtendrán los valores de los parámetros de la red comunes y necesarios para el resto de opciones, servirá también para preparar correctamente el entorno de entrenamiento, así como adquirir experiencia y conocimiento sobre como usar la API y entrenar correctamente a los agentes.

5.2.2. Aprendizaje por imitación

El segundo entrenamiento incorporará técnicas de imitación, de tal forma que los agentes aprovecharan conocimiento obtenido a partir de demos realizadas con la IA programada inicial.

5.2.3. Aprendizaje incorporando memoria

Finalmente, se hará uso de '*memoria*' para intentar mejorar los resultados. En este caso, *ML-Agents* ofrece dos alternativas: la primera es el propio aprendizaje LTSM y la segunda es incrementar el input de la red neuronal con *Stacked vectors*, de tal forma que se comporte como una matriz de vectores de inputs que se desplazan según se obtiene la información del entorno. Es por ello que, ante esta situación, se probarán ambas y se concluirá cual aporta mejores resultados.

5.3 Desarrollo en Unity

Ahora se verá como ha sido el desarrollo y la incorporación de la API de *ML-Agents* en el proyecto de Unity, tratándose del proceso más largo del proyecto. Así, primero se verán los pasos de la instalación de la propia API, para posteriormente mostrar las preparaciones y los cambios en el código original, dividiéndolo a su vez en secciones según su funcionalidad con el objetivo de clarificar toda la información lo mejor posible. Finalmente se explicarán por separado como ha sido cada entrenamiento.

5.3.1. Instalación de ML-Agents

Para la instalación ha sido necesarios diversos tutoriales y buscar en foros para corregir los diversos problemas que han ido apareciendo, no obstante, principalmente en [9] se encuentra todo detallado tratándose de la guía oficial.

Puesto que *ML-Agents* hace uso de librerías de python y lo utiliza como entorno donde realizar los cálculos para el entrenamiento, es necesario tener una versión instalada en el sistema. Para la versión actual a la realización de este proyecto de *ML-Agents* se recomienda 3.10.12 o superior, sin embargo, por diversas razones la API falla y ha sido necesario hacer uso de una versión anterior. Es por ello que, en este caso, se trabajará sobre python 3.9 para que todo funcione correctamente.

Una vez solucionado esto, basta con crear un entorno. Una alternativa muy recomendada es hacer uso de Anaconda, pero en nuestro caso nos sirve el propio módulo de python que permite crear entornos virtuales con el comando '*venv*'. Tras esto, se terminan de instalar todos los paquetes que sean necesarios, tanto en python como en Unity, y ya esta listo todo para funcionar.

5.3.2. Fichero .yaml con la configuración

Un elemento clave que irá apareciendo en todos los entrenamientos son los ficheros .yaml. Así, tratándose de un lenguaje centrado en representar datos de una forma fácilmente legible, contendrá toda la información necesaria sobre la red neuronal, de tal forma que siempre sufrirá cambios entre entrenamientos. Puede verse un ejemplo completo más adelante en la figura 5.5.

5.3.3. Comandos para iniciar el entrenamiento

Para iniciar el proceso de entrenamiento es necesario activar el entorno y hacer uso del comando '*mlagents-learn*' seguido de diversos parámetros:

- Una ruta que especifique el .yaml con la información de la red neuronal.
- '*–run-id*' seguido del identificador que se usará para la sesión de entrenamiento.
- '*–initialize-from*' seguido del identificador de una sesión anterior. Con esto se puede seguir entrenando una red pre-entrenada con el fin de obtener mejores resultados.

Existen más, pero estos son los más importantes de los que se usarán durante este proceso.

A continuación es necesario preparar el código ya existente para incorporar el uso de las redes, contenido del siguiente apartado.

5.3.4. Preparación del proyecto para el entrenamiento

Antes de comenzar con el proceso de entrenamiento es necesario realizar cambios y pruebas para integrar los nuevos comportamientos al proyecto. Como ya se ha presentado, el objetivo es añadir una nueva entrada en el código de *ModuloMovimiento* (1*i*) de tal forma que el resto quede inalterado. No obstante, en el resultado final, han sido inevitable multitud de ajustes para facilitar y ampliar las posibilidades a la hora de entrenar. Todo el código de esta nueva versión se puede ver en el apéndice C.

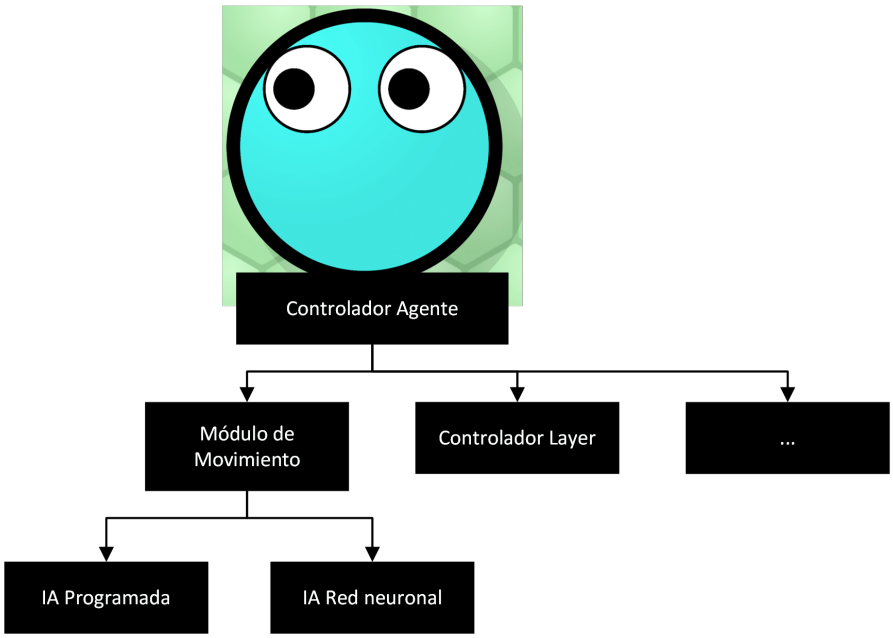


Figura 5.2: Estructura del agente incluyendo la IA de Red Neuronal

El principal paso ha sido crear un nuevo módulo para el agente, llamado *IARedNeuronal* (código en 1*i*), de tal forma que la nueva jerarquía de módulos sería, de forma simplificada, la que se puede ver en la imagen 5.2. La gran mayoría de ejemplos de proyectos similares que se encuentran en la web suelen comenzar a desarrollar la IA desde un comienzo, implementando toda la funcionalidad del agente en el propio módulo de la red neuronal. Es por esto que, como la implementación utilizada en este trabajo ha requerido una aproximación diferente, donde debe existir una comunicación entre los módulos ya existentes y el nuevo, solo la información sobre como funciona la librería y las bases teóricas han sido útiles en los documentos buscados.

5.3.5. Cambios realizados en el código

En este caso, el nuevo código no esta tan comentado y es por ello que se explicará en este apartado una introducción a como funciona la clase *Agent* que proporciona *ML-Agents*, así como que ha sido necesario implementar en este proyecto más en detalle. En caso de querer profundizar más en como funcionan los agentes, la información necesaria se encuentra en [7].

Es importante conocer el ciclo que pasará cada agente, siendo este el de: obtener las observaciones necesarias del entorno, tomar las decisiones pertinentes devolviendo un vector de acciones, y finalizando con el cálculo de las recompensas que indiquen al agente como de buenos son los resultados obtenidos. Así, para cada una de estas tareas, *ML-Agents* proporciona los siguientes métodos y funcionalidades a través de su clase *Agent*:

- Toma de decisiones:
  Para comunicar al agente que debe tomar una decisión se cuenta con el método *RequestDecision()*, que inicia todo el proceso para llevar a cabo el ciclo y calcular la acción pertinente. Además, aunque en este proyecto no se utilice, existe la clase *Decision Requester* que se encarga de llamar a este método de forma periódica.

- Observaciones:
  En cuanto a observaciones, existen dos tipos: vector y visuales. Las de vector se proporcionan a través del método *CollectObservations()* donde el usuario debe especificar los valores, mientras que para las visuales basta con indicar que elementos proporcionaran la información (cámaras o *Render Textures*). Además, *ML-Agents* cuenta con sus propios sensores, que pueden llegar a ser incluso *RayCasts*.

- Vectores de acciones:
  Los vectores de acciones contiene el resultado proporcionado por la red, y en *ML-Agents* pueden ser de tipo discreto o continuo. En cualquier caso, para solicitar dicho resultado se utiliza el método *AgentAction()*, aunque también existe *OnActionRecieved()* que se ejecuta automáticamente tras calcular una acción. Por último, el método *Heuristic()* permite indicar al usuario de forma "manual"la salida, funcionalidad útil para el entrenamiento por imitación como se verá en 5.3.7.

- Sistema de recompensas:
  Finalmente para indicar al agente como de bueno ha sido el resultado tras su acción, se cuenta con un sistema de recompensas donde -1 significa peor resultado y +1 mejor resultado. Es recomendable mantener este valor dentro de este rango para mantener la estabilidad, aunque no es obligatorio. Existen dos métodos para actualizarlo, siendo *AddReward()* para añadir un valor a la recompensa y *SetReward()* para cambiar el valor actual.

Por otro lado, existe una clase de propiedades llamada *Behaviour Parameters* con la información de la red neuronal. Dichas propiedades incluyen aspectos ya comentados, pero es interesante indicar cuales son y de que forma se utilizan:

- *Behavior Name*: Identificador de la red neuronal.

- *Vector Observation*: Indica el tamaño del vector de observaciones, así como la posibilidad de usar *Stacked Vectors*, que serviría para crear la matriz de vectores entrada en función del tiempo.

- *Vector Action*: Indica el tipo de resultado y, en caso de continuo el tamaño del vector, mientras que para discreto el número de ramas para marcar cuantas acciones discretas simultaneas existen.

- *Model*: Sirve para añadir un modelo ya entrenado.

- *Inference Device*: Dispositivo a usar para llevar a cabo la inferencia (CPU o GPU).

- *Behaviour Type*: Comportamiento del agente, que puede ser el de entrenamiento a través de python, heurístico para que el usuario indique las salidas o inferencia para utilizar el modelo proporcionado.

- *Team ID*: En caso de utilizar *Self-play* (explicado en el apartado 5.3.6), indica a que equipo pertenece el agente.

- *Use Child Sensors*: Indica si debe utilizar los sensores que tiene el agente.

- *Max Step* de *Agent*: Utilizado para indicar cuantas actualizaciones puede realizar el agente antes de reiniciarse.

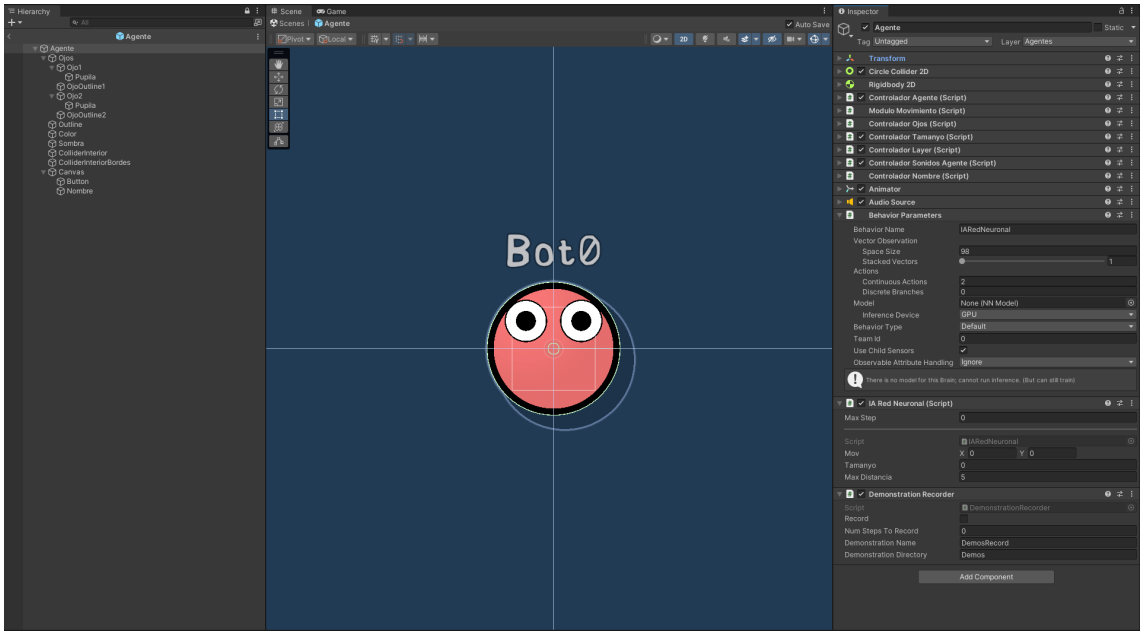


Figura 5.3: Nuevo prefab del agente en Unity

Con todo esto, en la imagen 5.3 se puede observar la adición de estas nuevas clases y los valores de los parámetros necesarios, así como otra clase llamada *Demonstration Recorder* que se explicará en el apartado 5.3.7. A continuación se tratará cada parte del ciclo por separado y la preparación del entorno para entrenar, explicando más en detalle como se ha conseguido cada una y las decisiones que se han tomado en el código. Aclarar que los resultados que se van a exponer se han obtenido tras realizar diversas pruebas e ir valorándolos hasta que han sido lo suficientemente adecuados, en el apartado 5.3.6 se explorará más este asunto.

Recolectar observaciones

Para el apartado de observaciones se encuentra el vector de *ObjetosDeRayCast* que el agente calcula en cada iteración. No obstante es necesario añadir algún input más y aplicar ciertas operaciones para normalizar y reducir la cantidad de valores que puede adoptar la entrada, con el fin de simplificar el problema. Por tanto, los valores que acaban recibiendo son el movimiento previo realizado y, por cada objeto observado:

- Un entero que indica si no se ha observado nada con 0, si se trata de una pared con 1, y de ser agente o alimento con 3 (normalmente se diferencia entre uno u otro, pero en este caso esta diferenciación no aporta información útil y se ha optado por tomarlos como uno sólo).
- El valor de la distancia normalizada a la que se ha observado el objeto. Para normalizar esta distancia se tiene en cuenta la máxima a la que puede ver el agente según su tamaño actual, valor pasado por la clase de *ControladorAgente*.
- El valor proporcional entre el tamaño del elemento observado y del agente, fruto de la división del primero entre el segundo. Además, para mejor clarificación del valor, se aplica unas operaciones de tal forma que el resultado final pasado sería: cuando se encuentra entre 0 y 0.8 se multiplica por 1.25 con el fin de revertir la diferencia proporcional consiguiendo que de 0 a 1 signifique que se puede eliminar, entre 0.8 y 1 mantiene un valor de -1 indicando que es neutro (ni se puede eliminar ni puede eliminar), y a partir de 1 el negativo de su valor indicando que es peligroso.

Si se compara, la clase *ObjetosDeRayCast* obtiene los valores brutos con información del entorno, pero para un mejor entrenamiento se ha considerado oportuno normalizar los valores de tal forma que se elimine todo aquello que no aporte utilidad (la mayoría de esto se llevan a cabo en el método *CollectObservations()* en 1*i*). Por un lado, la posición que mantiene el objeto observado se puede sustituir por la distancia y el *RayCast* que ha detectado el objeto, el tipo se ha reducido al mínimo necesario, y tanto el tamaño como la distancia se han normalizado siguiendo unas reglas que consiguen omitir el lugar y momento exacto de la partida consiguiendo entradas más genéricas para la red neuronal.

Además de todo esto, para terminar de ajustar los valores, y a diferencia de la versión original donde era irrelevante, en el momento de guardar los datos en el vector de *ObjetosDeRayCast* dentro de *ControladorDeAgente* se ha asegurado de mantener cierta consistencia, de tal forma que si el *ray cast* no ha detectado nada tanto el tamaño y la distancia sería igual a 0, y si detecta una pared el tamaño sería de -1.

Vector de acciones

En cuanto al vector de acciones, se hace uso de la función *OnActionReceived()*, de tal forma que tras realizar una llamada a *RequestDecision()* este actualiza el valor de la variable *mov* que usa *ModuloMovimiento* para pasar la información finalmente a *ControladorAgente*. Este vector esta compuesto por dos reales para indicar dirección x e y, y siempre se pasa normalizado para maximizar la velocidad del agente.

Sistema de recompensas

El sistema de recompensas es uno de los apartados más importantes ya que es el que guía al agente a la hora de aprender que acciones debe tomar en que situaciones. Además, también ha sido uno de los más complejos de diseñar debido a que es necesario valorar y tener multitud de factores en cuenta, sin embargo al final se ha conseguido algo que, sin ser perfecto y tener defectos, parece que ha aportado resultados suficientemente satisfactorios.

Para llevar a cabo esta tarea se utilizan los métodos ya mencionados de *AddReward()* y *SetReward()* en ciertas partes tanto del código de *ControladorAgente* como de *ModuloMovimiento*. En cuanto a los detalles, con el fin de que se comprenda lo mejor posible, se dividirá este sistema en diversas partes según su función dentro del mismo:

- Una pieza que aparece en el resto de partes es el sistema de peligro. Este se encarga de ajustar gran parte de las recompensas para que tengan en consideración la presencia de agentes hostiles en los alrededores. Así, al calcular los *ray casts*, simultáneamente se va acumulando por cada uno un valor que representa que tanto peligro se ha detectado. Concretamente, se valora tanto la distancia como el tamaño en caso de que se trate de un agente, y se divide por el número de *ray casts* totales, obteniendo tras la suma total un número entre 0 y 1, donde 0 significa seguro y 1 máximo peligro (todo el código de esta parte se encuentra en *ControladorAgente*).

- Por otro lado, es interesante que el agente siempre traté de buscar aumentar su tamaño, y para ello en *ModuloMovimiento* se ha elaborado un temporizador que, si detecta que el agente no ha consumido nada en un periodo, comienza a bajar su valor de recompensa linealmente con el tiempo hasta que sobrepase -1. Además, como se había adelantado, hace uso del control de peligro para acentuar el decrecimiento cuanto mayor peligro haya.

- La parte que indica al agente que progresa adecuadamente cuando consume alimentos o elimina a otros agentes se encuentra en *ControladorAgente*, pues cuando se procede a realizar el aumento de volumen, también se aumenta el valor de recompensa. En este caso se utiliza la misma relación proporcional entre el tamaño del elemento observado y del agente ya mencionada en el apartado de observaciones, pero solo se tiene en cuenta que tiene que ser el mínimo entre esta división y 0.1, asegurando que nunca se pierde interés por muy grande que sea el agente. Aquí también se tiene en cuenta el control de peligro, penalizando el resultado de este calculo cuanto mayor sea el valor de peligro.

- Finalmente, igualmente en *ControladorAgente*, cuando un agente es eliminado automáticamente se establece su valor de recompensa a -2.

El apartado del temporizador se ha programado para que se puedan modificar, con el objetivo de que durante el entrenamiento se pueda ajustar según las necesidades del momento.

Entorno de entrenamiento

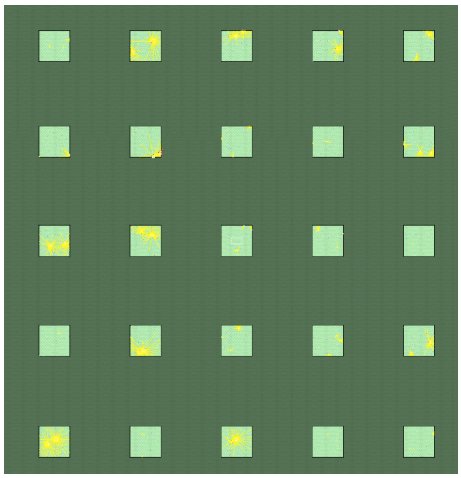


Figura 5.4: 25 partidas simultaneas en el entorno de entrenamiento

Fuera de lo relacionado con los agentes se encuentra el entorno (en la imagen 5.4 se puede observar cual ha sido a grandes rasgos el estado final del mismo), y es que para llevar a cabo un correcto entrenamiento ha sido necesario modificar el código de las clases que se encargan de controlar el mapa.

Partiendo de la escena original donde se desarrollan las partidas, primeramente ha sido necesario crear un *prefab* que permita duplicar libremente el escenario completo, y gracias a la estructura que sigue este no ha supuesto más complicaciones que crear un objeto padre que englobe todo y crearlo a partir de este. Con esto ya se puede elaborar un nuevo escenario que ejecute partidas simultáneamente, aunque es necesario realizar ajustes para que puedan durar indefinidamente.

A continuación se han modificado el código de *ControladorAgente* y *ControladorAgentes*, añadiendo los métodos de *ResetearEstado()* y *ResetearPartida()* respectivamente, cuya funcionalidad es la de retornar la partida que los ejecute a su estado original cuando fueron creados. Sin embargo, para que funcionase todo correctamente se han necesitado de pequeños cambios que inhabilitan cosas por un lado y creen otras por otro en los códigos de *ControladorAgentes* y *AlimentoSpawner* para mantener la correcta funcionalidad del conjunto.

Como punto adicional, aunque al final quedo en desuso debido a que no fue requerido lo que ofrece, se implementó la opción de ejecutar un tipo de escenario en el que un único agente con red neuronal competiría contra el resto de agentes con la inteligencia artificial programada. Básicamente, en *ControladorAgentes* se añadió una opción para indicar este comportamiento, de tal forma que se encargaba de que el agente 0 cumpliera este requisito, y en caso de ser eliminado reiniciaba el escenario. Esta era una solución ante los problemas iniciales de entrenamiento, pero fue reemplazado por otros métodos más rápidos y eficientes.

Con todas estas preparaciones ya realizadas, lo siguiente es iniciar el proceso de entrenamiento, donde se verán también cambios más específicos según hayan sido requeridos dentro del propio proceso.

5.3.6. Entrenamiento mediante aprendizaje por refuerzo

El proceso de entrenamiento esta compuesto principalmente de dos elementos: por un lado se encuentra el fichero .yaml con toda la información sobre la red neuronal, y por otro lado la configuración y funcionalidad preparada en el entorno de Unity.

Al tratarse del primer entrenamiento, aquí ha sido el momento de ir probando variantes y versiones incompletas de todas las partes del ciclo ya mencionadas en el apartado 5.3.5:

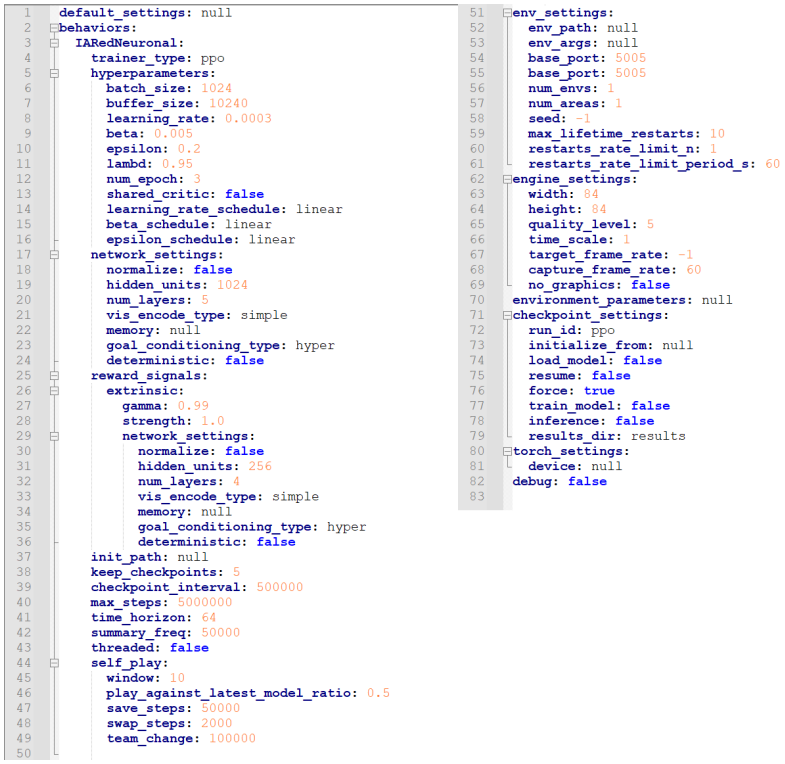
- En la parte de toma de decisiones, al comienzo se uso la clase *Decision Requester*, que terminó siendo descartada con el fin de que las decisiones se tomarán en los momentos adecuados (también fue necesario varias pruebas hasta dar con el lugar idóneo, pero realmente no habían demasiadas opciones).
- Las observaciones fue uno de los apartados que más tiempo y cambios necesitó, comenzando con una primera versión que pasaba directamente los datos que se obtenían de los *raycast* y obteniendo resultados para nada útiles, ya que los agentes tenían de alguna forma que aprender a jugar teniendo en cuenta demasiado el contexto, poco a poco se fueron aplicando normalizaciones y simplificaciones hasta obtener la versión actual.
- El vector de acciones fue bastante directo, únicamente siendo necesario dar con aplicar la normalización al vector de salida.
- Finalmente el apartado de recompensas que, al tratarse del que guía al agente, es el más importante y difícil de establecer de todo el ciclo. En las primeras pruebas era algo simple, donde el agente únicamente recibía recompensas positivas si consumía y negativas si era eliminado. Claramente esto no era suficiente y progresivamente se fueron añadiendo a esto nuevos conceptos e implementaciones que hacían al sistema más complejo. Por un lado, primeramente el temporizador que obliga al agente a continuamente buscar algo que consumir, luego se incorporaron las ponderaciones a las recompensas por consumir y por último el sistema de peligro para que los agentes también valorarán la presencias hostiles. Durante el proceso de averiguar estas características e implementarlas, existieron variantes de las mismas que incluían imperfecciones que se fueron corrigiendo, como que las recompensas no se encontrasen entre -1 y 1.

Después de todo, se pudo llegar a un entorno de Unity bastante estable el cual es el ya descrito en 5.3.5 y del que partirán todos los entrenamientos.

Continuando por el fichero .yaml, sin contar los eliminados durante el periodo de prueba, todos se encuentran en 9*a*, donde cada uno de ellos esta preparado para cada tipo de entrenamiento:

- configuracionTraining.yaml: Son las configuraciones básicas.
- configuracionTrainingImitacion.yaml: Son las configuraciones preparadas para usarse en el entrenamiento que incorpore el bloque de imitación.
- configuracionTrainingStacked.yaml: Son las configuraciones preparadas para usarse con el primer tipo de memoria donde se apilan vectores incrementando la entrada.
- configuracionTrainingLSTM.yaml: Son las configuraciones preparadas para usarse con el entrenamiento que incorpore LSTM.
- configurationOnlyOne: Fue la configuración pensada para usarse en caso de activar el escenario ya presentado donde existe un único agente con red neuronal.

El resto se trata de configuraciones utilizadas durante las pruebas que no aportan datos demasiado relevantes, mientras que el fichero *Prompts* sirve de ayuda para mantener el último comando ejecutado para iniciar el entrenamiento. Esto se debe a que, como se verá más adelante, el proceso de entrenamiento se vuelve bastante complejo y es fácil equivocarse a la hora de ejecutar estos comandos.



```yaml
default_settings: null
behaviors:
  IARedNeuronal:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 10240
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      shared_critic: false
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 1024
      num_layers: 5
      vis_encode_type: simple
      memory: null
      goal_conditioning_type: hyper
      deterministic: false
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
        network_settings:
          normalize: false
          hidden_units: 256
          num_layers: 4
          vis_encode_type: simple
          memory: null
          goal_conditioning_type: hyper
          deterministic: false
    init_path: null
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 5000000
    time_horizon: 64
    summary_freq: 50000
    threaded: false
    self_play:
      window: 10
      play_against_latest_model_ratio: 0.5
      save_steps: 50000
      swap_steps: 2000
      team_change: 100000
env_settings:
  env_path: null
  env_args: null
  base_port: 5005
  base_port: 5005
  num_envs: 1
  num_areas: 1
  seed: -1
  max_lifetime_restarts: 10
  restarts_rate_limit_n: 1
  restarts_rate_limit_period_s: 60
engine_settings:
  width: 84
  height: 84
  quality_level: 5
  time_scale: 1
  target_frame_rate: -1
  capture_frame_rate: 60
  no_graphics: false
environment_parameters: null
checkpoint_settings:
  run_id: ppo
  initialize_from: null
  load_model: false
  resume: false
  force: true
  train_model: false
  inference: false
  results_dir: results
torch_settings:
  device: null
debug: false
```

Figura 5.5: Configuración de entrenamiento básica

En la imagen 5.5 se cuenta con toda la información de los parámetros básicos para poder comenzar a entrenar, donde se encuentra lo siguiente:

- El apartado de *hyperparameters* donde se incluyen los valores que dirigen el proceso de entrenamiento (por ejemplo, *batch_size* indica cuantos inputs son necesarios para realizar la actualización de pesos, el *learning_rate* indica la intensidad a la que se cambiarán dichos pesos, etc.) no ha necesitado cambios con respecto a la versión original. Sin embargo, diversas pruebas en las que se aumentaba el valor de *learning_rate* acababan por fallar, de tal forma que el agente se movía siempre en la misma dirección.

- En *network_settings* únicamente se han modficado los valores de *num_layers* y *hidden_units* , correspondiendo al número de capas y al número de nodos que contiene cada una respectivamente. En concreto, ha sido necesario aumentar su valor hasta donde se ha mantenido estable (también ocurría el error comentado en la sección anterior si eran valores grandes) debido a la complejidad del entorno a la que se someten los agentes.

- *Reward_signals* no se ha modificado en este apartado, pero adelantar que, ya que sí será útil sobre todo en imitación, es el encargado de indicar que tipo de valores tendrá en cuenta la recompensa del agente.

- Un apartado que ha sido necesario añadir es el de *self_play*, que esta relacionado con la variable *TeamID* en el *Agent* de Unity, siendo el encargado de cambiar la forma de entrenar teniendo en cuenta que los agentes compiten entre ellos. Este es uno de los puntos más importantes a tener en cuenta, puesto que si no se incluye los agentes se comportan de forma cooperativa generando resultados indeseados tales como dirigirse a una esquina con el fin de que uno de ellos aumente al máximo y gane.

- Finalmente, como dato interesante, las pruebas parecen indicar que dependiendo de la escala de tiempo que se use, los agentes aprenden a jugar en base a dicha escala. Esto significa que, aunque se pueda ver como buena idea entrenar con una escala de tiempo superior a uno para reducir tiempos de entrenamiento, esto dará problemas cuando la red se use en escala normal creando una situación el la que los agentes se mueven de forma caótica pese a que durante dicho entrenamiento todo funcionaba correctamente (corresponde al valor de *time_scale* en *engine_settings*).

Para más información sobre el funcionamiento del *self_play* en [6], y para más información sobre el fichero de configuración visitar [4].

Con todo estos preparativos listos puede dar comienzo el proceso de entrenamiento conectando Unity con python y ejecutando los comandos necesarios. Sin embargo, las pruebas iniciales donde se entrena a lo bruto indican que esta no es la forma adecuada de proceder, ya que los agentes muestran indicios de ser incapaces de jugar correctamente, por lo menos en el tiempo que se les deja entrenar. La razón principal por la que puede darse este suceso es que el entorno sea demasiado complejo, y es que si se tiene en cuenta que los agentes deben ser capaces de realizar acciones para consumir otros agentes o alimentos mientras esquivan a otros en un mapa cerrado, todo ello a partir de información bastante compleja de tipos, tamaños y distancias, se puede concluir que dicha complejidad existe y que a los agentes les cuesta adaptarse correctamente.

Ante este problema, una solución que parece ser suficientemente adecuada ha sido la de realizar un proceso de entrenamiento progresivo y adaptativo. Esto implica llevar a cabo múltiples ejecuciones de entrenamiento partiendo siempre de la versión anterior, de tal forma que al agente se le van planteando las cosas poco a poco. Más concretamente y a lo largo de 6 ejecuciones:

- Se comenzaba con un número de agentes reducido de 4 o menos, aumentándolo progresivamente por cada ejecución hasta alcanzar entre 20 y 30, con el fin de que el agente primeramente aprenda a alimentarse para posteriormente, según va avanzando, comience a huir y valorar el peligro.
- El número de escenarios activos comienza en los 25 y se van reduciendo en función del número de agentes para mantener el rendimiento lo más estable posible.
- Se cambia el parámetro que controla la penalización en la recompensa del agente cuando pasa tiempo sin variar de tamaño, comenzando entre 3 y 2 segundos, y acabando en 0. De esta forma también se vuelve progresivo como de exigente es que los agentes realizen acciones.

Finalmente, con esta metodología ha sido posible obtener la red neuronal entrenada únicamente utilizando las bases del entrenamiento por refuerzo.

5.3.7. Entrenamiento incorporando imitación

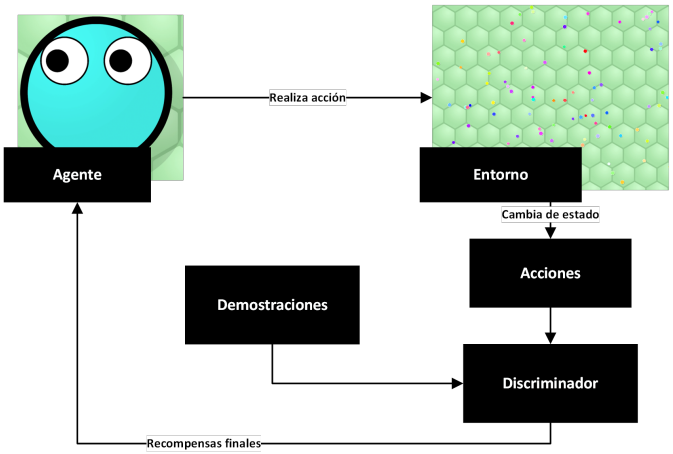


Figura 5.6: Esquema del entrenamiento por imitación

En la imagen 5.6 se puede observar un esquema del funcionamiento del agente cuando se añade *Generative Adversarial Imitation Learning* (*GAIL*), que se basa en el concepto de imitar comportamientos. Para ello se añade un nuevo apartado al ciclo del agente en el que se comparan las acciones y resultados obtenidos (discriminador) con un conjunto de demostraciones proporcionadas, alterando las recompensas que finalmente recibe el agente. Además de esto, también se añadirá *Behavioral Cloning* que modifica la salida de la red para que imite exactamente a las demostraciones, pudiéndose ajustar la intensidad de esta aproximación.

En cuanto a la implementación en Unity, ha sido necesario recolectar las demostraciones a través del uso del método *Heuristic()*. Para ello, tras ejecutar con normalidad el paso de la inteligencia artificial programada cuando el agente se mueva, se actualiza el valor de *mov* de *IARedNeuronal* a este resultado y se le pide que tome una decisión al agente que, asegurando que tiene marcado el *Behavoir Type* como heurístico, tomará dicho *mov* como salida manual. Con esto, algunos pequeños cambios para mantener la estabilidad y configurar la clase de *Demonstration Recorder* vista en la figura 5.3 que se encarga de de las funcionalidades dentro de *ML-Agents*, basta con activar los escenarios deseados y inicar la ejecución. Al finalizar, se encontrará en el directorio indicado (en nuestro caso en *9b*) todos los ficheros de demostración. Para este proyecto se llevo a cabo una ejecución de un escenario con 30 agentes.

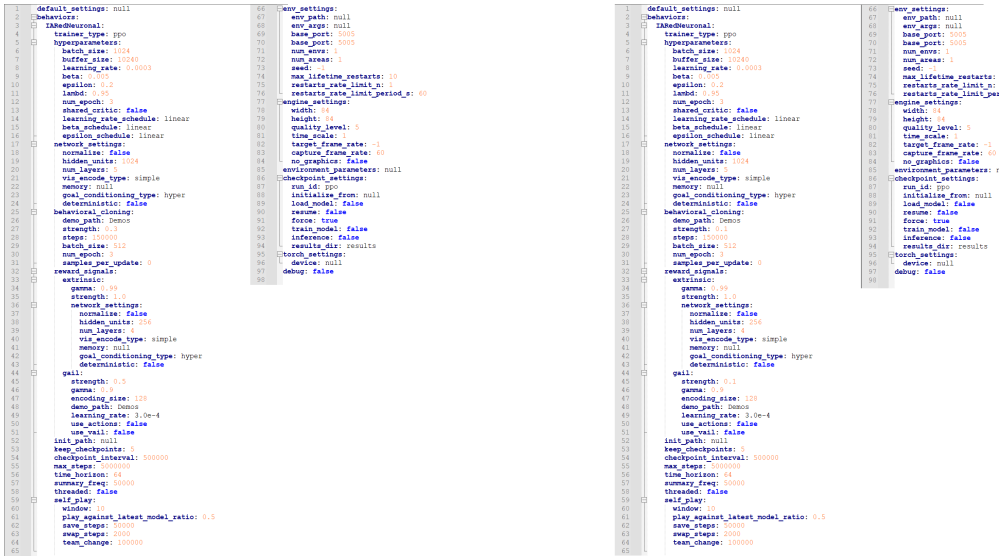


Figura 5.7: Configuraciones para el entrenamiento con imitación

En principio el resto del código se mantiene igual al del entrenamiento por refuerzo básico, sin embargo aun quedaría ajustar el .yaml con la información de la red neuronal. Para este modelo se contará con dos versiones como se puede observar en la imagen 5.7, la primera para los entrenamientos iniciales y la segunda para los finales. A diferencia de la configuración original se han añadido dos apartados: *behavioral_cloning* y *gail* dentro de *reward_signals*. Ambos se encargan de activar y configurar las funcionalidades respectivas dentro de la red. En cuantos a sus parámetros, solo remarcar dos:

- *demo_path*: Directorio con todas las demostraciones.
- *strength*: Indica la intensidad a la que se aplicará cada función, por un lado en *behavioral_cloning* con respecto imitar exactamente a las demostraciones, mientras que en *gail* ponderará esta recompensa con el resto activas.

Para llevar a cabo el entrenamiento se ha reducido ambos valores de *strength* de la segunda configuración con respecto a la primera con el fin de que al comienzo si que aprenda a copiar más a las demostraciones, pero que en los finales traté de dar con estrategias propias. Así, con esto y siguiendo la misma metodología progresiva que en el entrenamiento básico, se consigue la red neuronal entrenada con imitación.

5.3.8. Entrenamiento incorporando un sistema de memoria

Como se comentó en el apartado de análisis de alternativas 5.2, existen al menos dos formas para dotar a los agentes de memoria: mediante la ampliación de los datos de entrada convirtiéndolo en una matriz de vectores de observaciones, o indicando el uso de LSTM (*"Long Short-Term Memory"*).

En el caso de hacer uso de la primera opción basta con modificar el parámetro de *Stacked Vectors* de la clase *Behavior Parameters*, habiendo indicado para este proyecto el uso de 25 vectores. De esta forma, la API de *ML-Agents* se encarga automáticamente ir acumulando vectores de forma cíclica según se pasan los datos de las observaciones. Para comenzar el entrenamiento no son necesarios más cambios, y simplemente a partir de la configuración base y siguiendo la misma metodología progresiva podemos obtener resultados. Sin embargo, debido a los comportamientos generados durante el proceso, se ha optado por entrenar siguiendo dos pequeñas variantes: una con 512 *hidden_units* y otra con 1024.

```
17    network_settings:
18      normalize: false
19      hidden_units: 512
20      num_layers: 5
21      vis_encode_type: simple
22      memory:
23        sequence_length: 64
24        memory_size: 256
25      goal_conditioning_type: hyper
26      deterministic: false
```

Figura 5.8: Configuración nueva para LSTM

En cuanto al uso de LSTM, basta con incluir la información de *memory*, como se puede ver en la figura 5.8, para activar esta funcionalidad. Este tipo de memoria es bastante interesante al ser selectiva, lo que significa que, a diferencia del anterior donde el agente básicamente interpretaba toda la información como útil (complicando y alargando el entrenamiento), se entrena un componente especializado en decidir que información es importante en cada momento. Este proceso conlleva que el entrenamiento sea más complejo, y es por esto que se trabajará con una red de 512 nodos por capa oculta. De igual forma, sin más cambios y con el mismo procedimiento de siempre podemos obtener una red neuronal entrenada a partir de esta idea.

Al final, los resultados muestran que los agente entrenados con *Stacked vectors* desempeñan de una forma deficiente. Una posible explicación es la complejidad del problema, y es que, como se ha descrito, han sido necesarios metodologías que ayuden a mitigar esto, y en este tipo de entrenamiento esta complejidad aumenta considerablemente debido a como de grande se vuelve el vector de entrada. Otra forma de entender porque no funciona correctamente es interpretando que para tomar una decisión toma en cuenta información que no aporta nada útil, al contrario que LSTM que es más selectiva. Es por todo esto que en las comparaciones únicamente se tendrá en cuenta LSTM.

CAPÍTULO 6
Comparación de los resultados obtenidos

En este capítulo se realizará una comparación de todas las redes entrenadas a lo largo del desarrollo del proyecto. Para ello, primeramente se indicará cuáles serán las métricas concretas que se utilizarán, para posteriormente iniciar dicha comparación mostrando de antemano todos los resultados obtenidos.

6.1 Métricas utilizadas

Principalmente se contará con 4 métricas:

- El tiempo necesitado para entrenar la red, donde se calculará aproximadamente cuanto ha estado entrenando activamente en Unity eliminando posibles pausas.
- El esfuerzo necesario para llevar a cabo el entrenamiento, donde se incluyen todas las tareas a partir de tener la red al menos funcional.
- Gráficas que se pueden visualizar a través de *tensorboard*, y que puede ayudar a averiguar cual a sido el desarrollo del entrenamiento. Concretamente una para la evolución de la recompensa y otra para la evolución de la duración de los episodios de los agentes[1].
- Una valoración relativamente subjetiva sobre como desempeñan finalmente los agentes cuando juegan en partida, sustentada sobre una serie de datos obtenidos a través de realizar 100 partidas de 30 segundos con 20 agentes. Para ser exactos, esos datos serán:
  - Tiempo se supervivencia promedio (cuanto mayor mejor y se trabaja en segundos).
  - Número de consumiciones promedio (cuanto mayor mejor)
  - Tamaño promedio (cuanto mayor mejor).
  - Agentes Vivos (cuanto mayor mejor).
  - Mínimo, máximo y media de tiempo sin consumir promedio (cuanto menor mejor y se trabaja en segundos).

[1] La duración de los episodios no se ha tratado hasta el momento, pero basta con conocer que acaban cuando el agente es eliminado y que, por tanto, la gráfica informa sobre la supervivencia de los mismos.

Los datos de la IA programada se pueden ver en 6.1, y se debe tener en cuenta que se han valorado tomando como objetivo la supervivencia del agente, así como que al tratarse de partidas tan cortas, una mínima diferencia tiene bastante peso.

En cualquier caso y para cualquier punto, pese a existir datos totalmente objetivos, se aportará una parte relacionada con la experiencia que se ha tenido y las consideraciones que puedan ser relevantes.

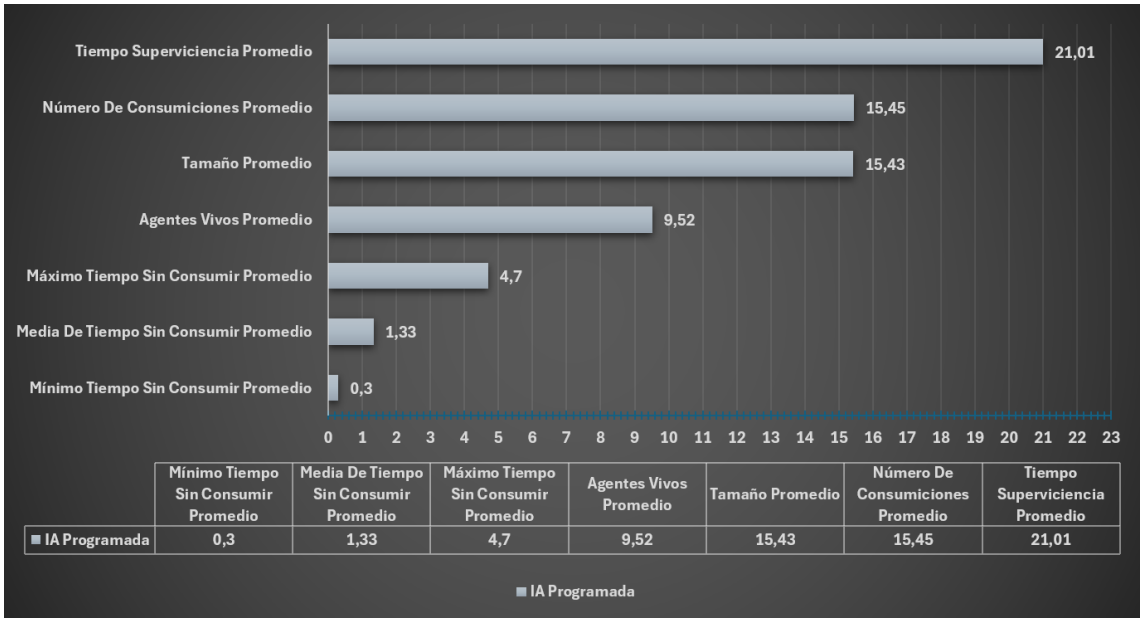


Figura 6.1: Datos obtenidos de la IA programada.

6.2 Resultados obtenidos

Para realizar las comparaciones, primero se presentarán todos los puntos nombrados para cada red por separado, para seguidamente acabar con una puesta en común y las conclusiones finales. Una cosa a tener en cuenta es que en el apartado de valoración subjetiva, al no poderse comparar entre ellos previamente al apartado de comparación, se tomará los datos de la IA programada como referencia para todos los casos.

6.2.1. Entrenamiento por refuerzo

Tiempo

Este entrenamiento esta compuesto por 6 iteraciones con duraciones de:

1.67 h	1.57 h	1.68 h	1.97 h	2.34 h	9.41 h

Tabla 6.1: Tiempos del entrenamiento por refuerzo.

En total, 18.64 horas de entrenamiento.

Esfuerzo necesario

Para este entrenamiento únicamente ha sido necesario ajustar parámetros y escenarios con el objetivo de realizarlo siguiendo la metodología progresiva ya explicada al final del apartado 5.3.6. Así, se puede considerar que se ha necesitado un esfuerzo no demasiado elevado.

Datos de *Tensorboard*

En las gráficas 6.2 y 6.3 se puede ver la evolución tanto de la recompensa como de la duración de los episodios de los agentes respectivamente, donde cada línea representa cada una de las iteraciones. Además, para comprender correctamente su significado, es importante valorar que entre iteraciones han habido cambios en el entorno. Así:

- En las 4 primeras existen pocos agentes compitiendo, de tal forma que solamente se centran en buscar alimentos. Es por ello que se puede ver a lo largo de las 3 iniciales una pequeña mejora en la recompensa, mientras que la duración se mantiene estable, pero en la cuarta, pese a no mejorar la recompensa, sí que mejora la duración indicando capacidad de supervivencia.
- En cuanto se pasa a escenarios con más agentes, baja mucho su rendimiento y no son capaces de mejorar considerablemente.

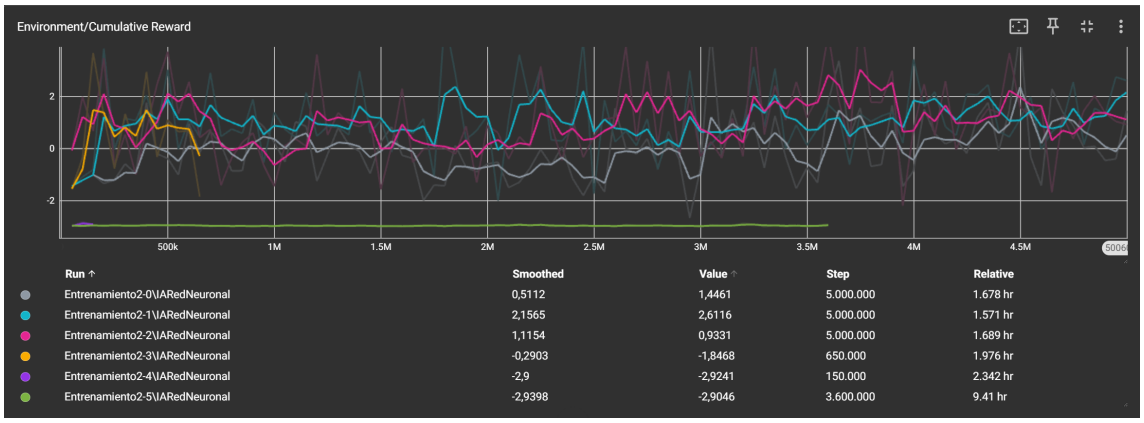


Figura 6.2: Gráfica de la evolución de la recompensa acumulada en el entrenamiento por refuerzo.

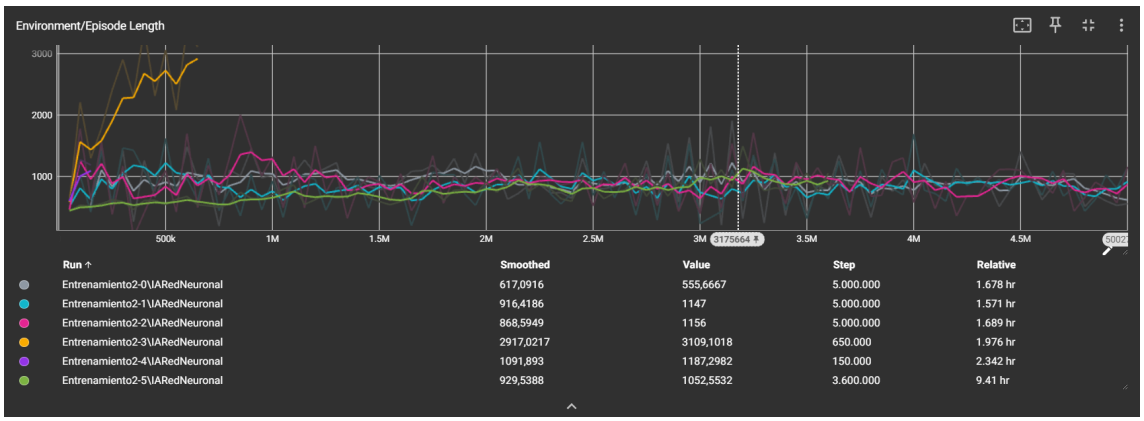
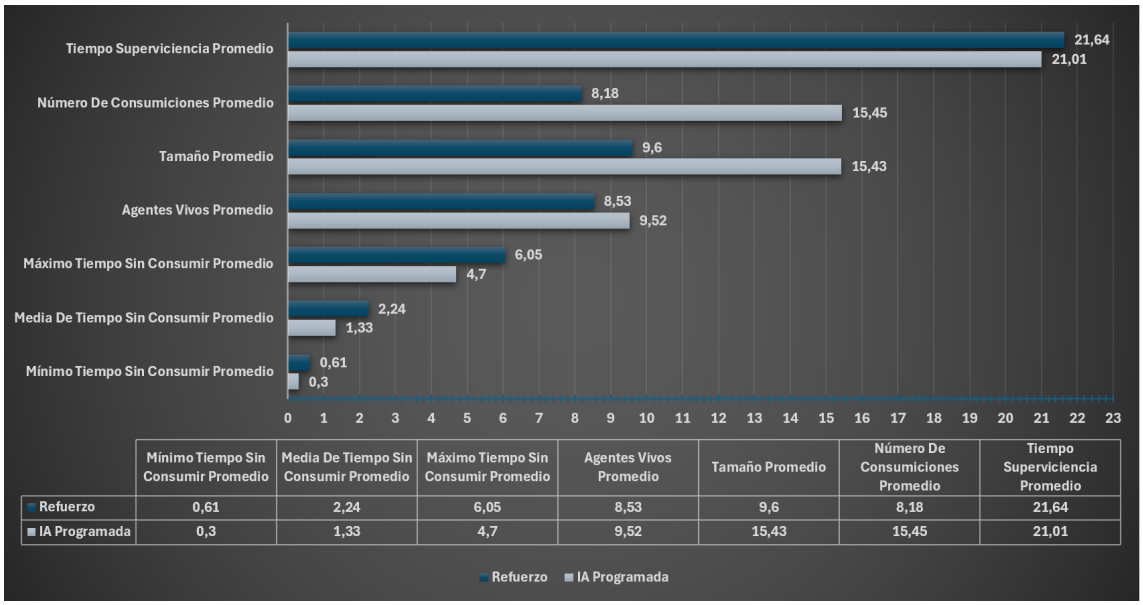


Figura 6.3: Gráfica de la evolución de la duración de los episodios en el entrenamiento por refuerzo.

Valoración subjetiva



	Mínimo Tiempo Sin Consumir Promedio	Media De Tiempo Sin Consumir Promedio	Máximo Tiempo Sin Consumir Promedio	Agentes Vivos Promedio	Tamaño Promedio	Número De Consumiciones Promedio	Tiempo Superviciencia Promedio
Refuerzo	0,61	2,24	6,05	8,53	9,6	8,18	21,64
IA Programada	0,3	1,33	4,7	9,52	15,43	15,45	21,01

Figura 6.4: Gráfica de barras para comparar refuerzo con la IA programada.

Tal y como indican las gráficas de *Tensorboard*, el comportamiento de los agentes no es el mejor. Cuando se encuentran en medio del escenario y exceptuando unos pocos casos en los que no se deciden, normalmente saben buscar alimentos correctamente. No obstante, no tienen la capacidad de huir correctamente en muchas ocasiones y además no evitan muchas veces las paredes, quedándose inmóviles contra las mismas.

Por otro lado, en la comparación que se puede ver en 6.4 ocurre un suceso anómalo bastante curioso, y es que en casi todos los campos rinde considerablemente peor a la IA programada, pero en el caso del tiempo de supervivencia lo supera por poco y en agentes vivos parece rendir mejor que en otros aspectos. Esto se debe al problema de que los agentes no sepan evitar paredes, llevando a que no sean capaces de eliminarse entre ellos, alterando los resultados y la interpretación de estos campos.

6.2.2. Entrenamiento incorporando imitación

Tiempo

En este caso esta compuesto por 4 iteraciones con duraciones de:

4.72 h	5.93 h	6.65 h	4.47 h

Tabla 6.2: Tiempos del entrenamiento con imitación.

En total 21.77 horas de entrenamiento. Sin embargo, para este caso en concreto, es interesante remarcar que en mucho menos tiempo los agentes eran capaces de jugar a un nivel aceptable, pero se prosiguió con el fin de ver hasta donde podían llegar. Esto se puede comprobar haciendo uso de las versiones intermedias de las redes, donde se ve que el comportamiento es muy similar al del final.

Esfuerzo necesario

Para este entrenamiento ha sido necesario realizar los preparativos para conseguir los datos que usarán posteriormente los agentes, incluyendo el propio proceso de grabación. Además, durante las iteraciones se han necesitado cambios extra relacionados con el funcionamiento de la imitación. Todo esto implica un esfuerzo considerablemente elevado.

Datos de *Tensorboard*

Igualmente se cuenta con las gráficas 6.5 y 6.6 cuyo significado se mantiene igual al anterior. En este caso, la evolución del número de agentes es tal que, durante las tres primeras iteraciones se centran más en enseñar la fase de alimentación mientras hay un aumento en el número de agentes progresivo, para por fin en la cuarta crear un escenario mucho más competitivo y con más libertad. Al final, en las gráficas ve reflejado que:

- En ambas se puede observar como los agentes se adaptan correctamente durante las dos primeras iteraciones. A partir de la tercera, la capacidad de supervivencia se mantiene e incluso se supera, creando que la recompensa global disminuya. Esto sucede porque, si de por sí una mayor cantidad de agentes implica una menor cantidad de comida del entorno para cada uno y por tanto menos recompensa, si se añade que no se eliminan entre ellos entonces la recompensa empeora considerablemente.
- Finalmente la última iteración parece confirmar este planteamiento, pues la duración de los episodios no tarda en alcanzar un punto elevado, conllevando que la recompensa global disminuya rápidamente también.

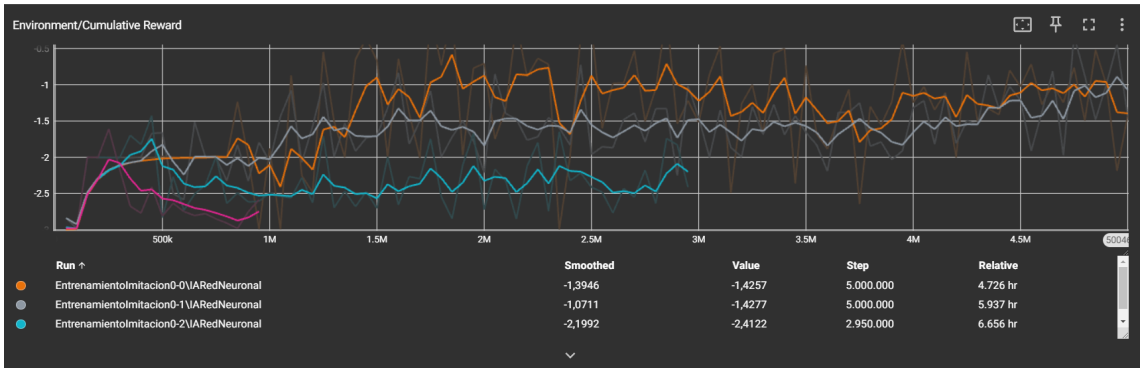


Figura 6.5: Gráfica de la evolución de la recompensa acumulada en el entrenamiento con imitación.

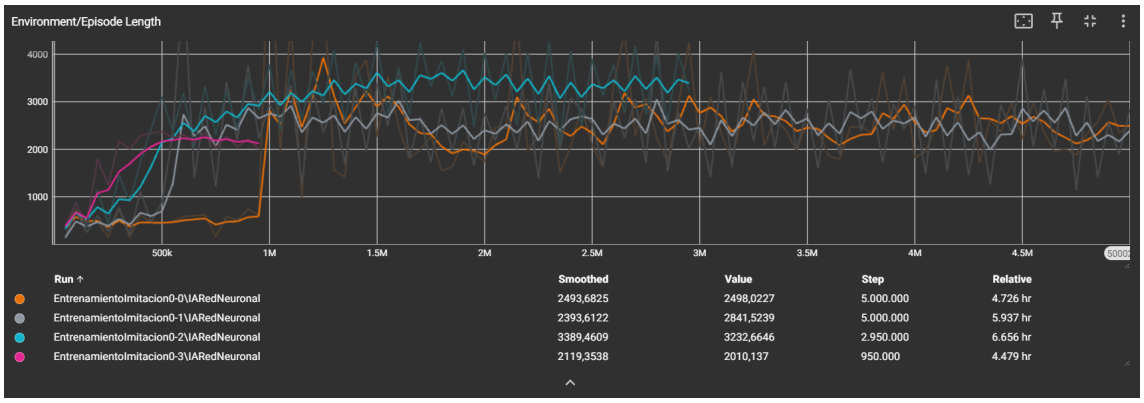


Figura 6.6: Gráfica de la evolución de la duración de los episodios en el entrenamiento con imitación.

Valoración subjetiva

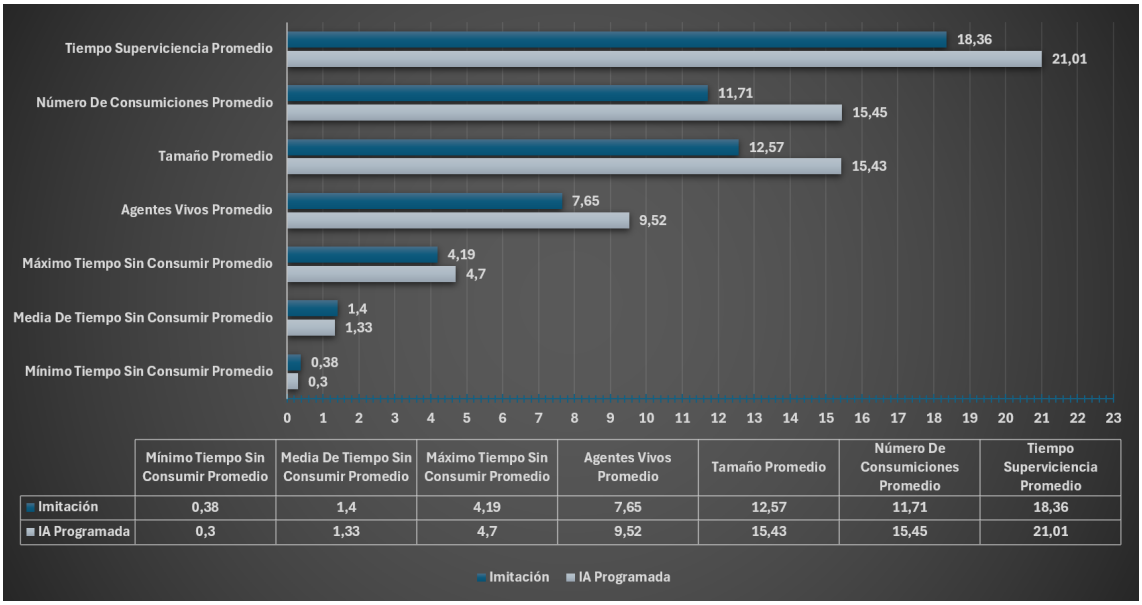


Figura 6.7: Gráfica de barras para comparar imitación con la IA programada.

En este caso, el rendimiento de los agentes es bastante bueno acercándose levemente al nivel de la IA programada. Prácticamente nunca dudan a la hora de buscar alimentos y además de forma muy rápida, saben esquivar normalmente a otros agentes y, pese a quedarse atascados en ciertas ocasiones, suelen ser capaces de salir de esa situación.

Todo esto se puede confirmar gracias a la gráfica en 6.7, donde se puede observar que, pese a rendir peor en general, se acerca en cierta mediada a la IA programada. De hecho, el tiempo máximo sin consumir lo mejora por poco, aunque cabe remarcar que existe la posibilidad de que esto se deba a que los agentes no escapan tan bien, tardando menos en eliminarse entre ellos.

6.2.3. Entrenamiento incorporando LSTM

Tiempo

En este caso esta compuesto por 4 iteraciones con duraciones de:

1.77 h	6.24 h	17.10 h	1.71 h

Tabla 6.3: Tiempos del entrenamiento con LSTM.

En total 26.82 horas de entrenamiento.

Esfuerzo necesario

A parte del entrenamiento progresivo llevado a cabo en el todos los entrenamientos, únicamente es necesario realizar cambios en el fichero .yaml de la configuración. Por tanto, se puede afirmar que no se ha necesitado casi esfuerzo extra.

Datos de *Tensorboard*

Como en el resto de entrenamientos, se comienza por pocos agentes durante los 2 primeros entrenamientos, para aumentar más en el tercero y acabar con un cuarto totalmente competitivo. En este caso, este último entrenamiento tiene poca duración porque el rendimiento en las partidas parecía mantenerse bastante estable, siendo innecesario proseguir. En las gráficas se puede observar como:

- Las dos primeras iteraciones muestran como los agentes simplemente mejoran tanto en recompensas como en duración de episodios, de tal forma que al comenzar la iteración comienzan por debajo y enseguida alcanzan un buen nivel.
- La tercera, aunque sigue el mismo patrón, queda algo inferior a las dos anteriores en recompensa, mientras que la duración si parece ir correctamente.
- Pese a que la última iteración sea algo corta, se puede observar que aproximadamente iba a tender a resultados similares.

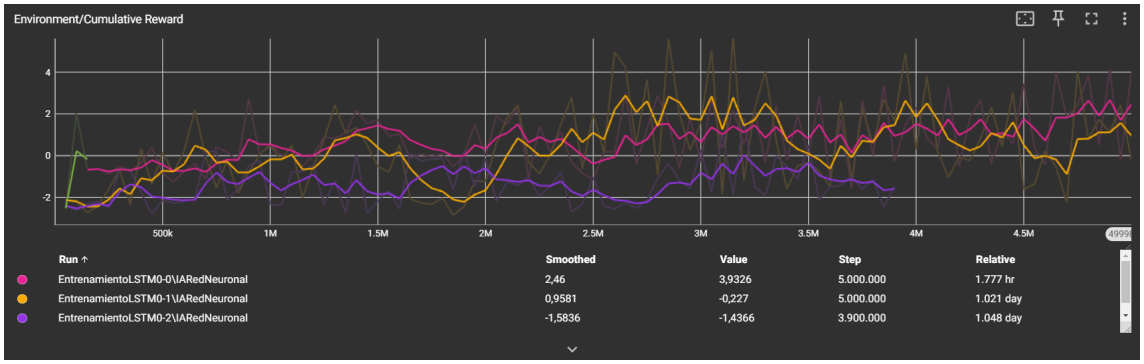


Figura 6.8: Gráfica de la evolución de la recompensa acumulada en el entrenamiento con LSTM.

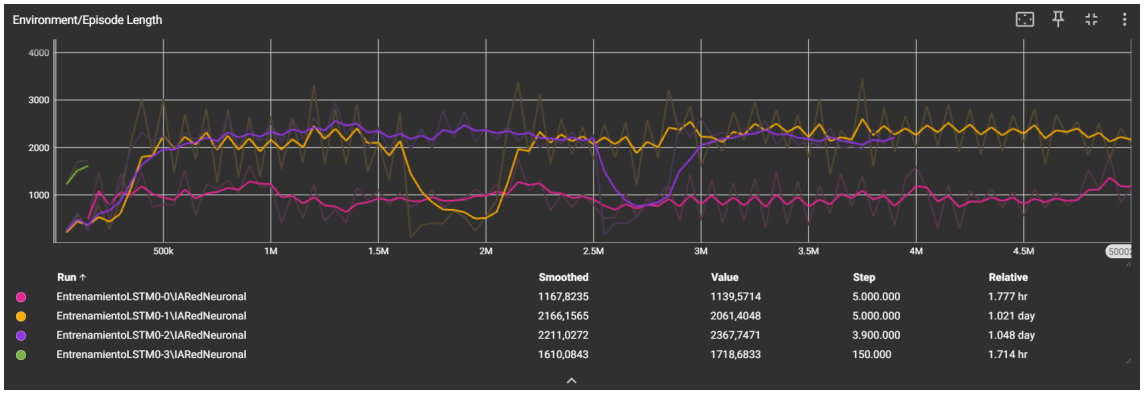


Figura 6.9: Gráfica de la evolución de la duración de los episodios en el entrenamiento con LSTM.

Valoración subjetiva

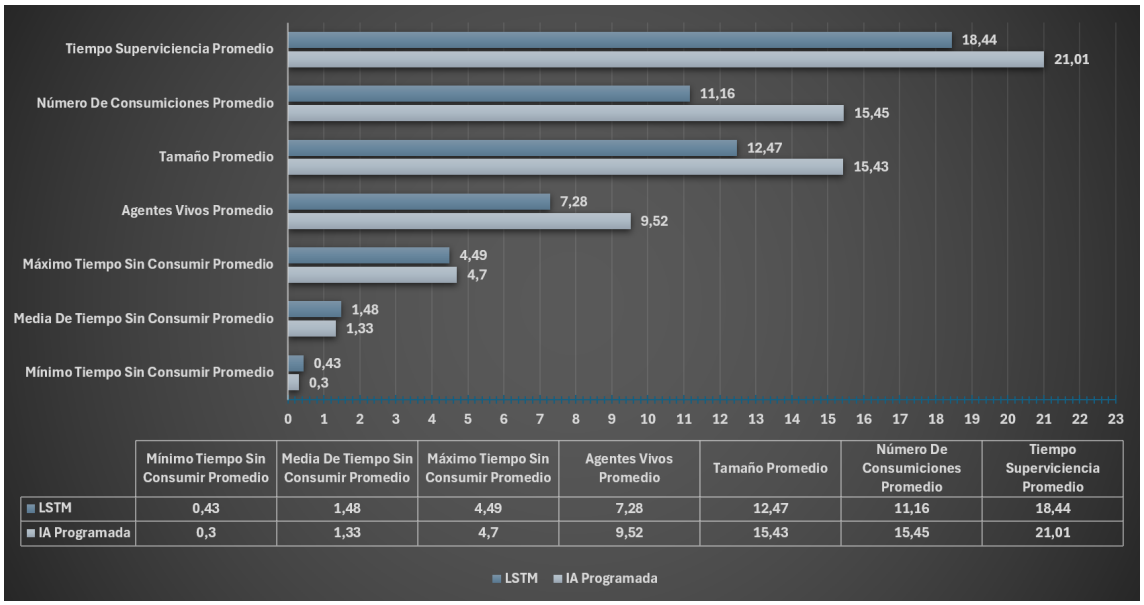


Figura 6.10: Gráfica de barras para comparar LSTM con la IA programada.

Finalmente, el desempeño de estos agentes ha sido más que aceptable, siendo estos capaces de decidir a la hora de consumir alimentos rápidamente, esquivar a otros agentes en algunas ocasiones y, a pesar de también engancharse en las paredes, muchas veces son capaces de salir de esta situación.

Nuevamente, la gráfica 6.10 confirma el correcto funcionamiento del modelo puesto que otra vez se acerca poco a poco a la IA programada. Además, vuelve a ocurrir la misma situación vista en el caso de imitación, donde se mejora el máximo tiempo sin consumir.

6.3 Comparaciones

Una vez presentados todos los datos obtenidos se puede iniciar la comparación entre ellos para sacar conclusiones e ideas interesantes.

18.64 h	21.77 h	26.82 h

Tabla 6.4: Tiempos totales de los entrenamientos por refuerzo, con imitación y con LSTM respectivamente.

En cuanto a tiempo se refiere y teniendo en cuenta la diferencia de calidad entre los agentes, es posible afirmar que el entrenamiento con imitación ha sido el que mejores resultados ha aportado en la menor cantidad de tiempo, puesto que pese a ser el segundo más largo, la segunda o tercera iteración ya eran suficientes como para superar como mínimo al entrenamiento por refuerzo. Por otro lado, incorporando LSTM es el que, debido a su complejidad, ha requerido de más tiempo, aunque también ha proporcionado muy buenos resultados. Finalmente, por refuerzo ha sido bastante rápido aun siendo 6 iteraciones, pero los resultados no son los mejores, lo cual puede implicar que hubiera sido necesario más tiempo de entrenamiento creando más iteraciones.

Por otro lado, el de imitación ha sido por contra el que más esfuerzo ha necesito con diferencia, ya que se puede tener en cuenta que, además de todo lo mencionado en los resultados, requiere de la creación de algún elemento para grabar las demostraciones, o incluso de grabarlas manualmente. Esto último no se ha valorado en los resultados ya que se supone que todos parten de la misma base, pero es interesante también tomar en consideración que entrenamiento ha precisado que elementos extra. Por otra parte, tanto por refuerzo y con LSTM han necesitado pocos cambios para poder llevarse a cabo.

En lo que se refiere a los datos de *Tensorboard* y el desempeño final de los agentes solo cabe remarcar como tanto en imitación y LSTM sí que se puede observar un progreso de aprendizaje por parte de los mismos, donde existe cierto equilibrio entre su supervivencia y su capacidad para consumir y eliminar a otros, mientras que en el de refuerzo no esta tan claro, siendo un aprendizaje más bien caótico. Todo esto se ve reflejado en las partidas, donde, a diferencia del entrenado únicamente por refuerzo, LSTM y imitación, con este último algo por encima, pueden llegar a ser utilizadas en el juego final.

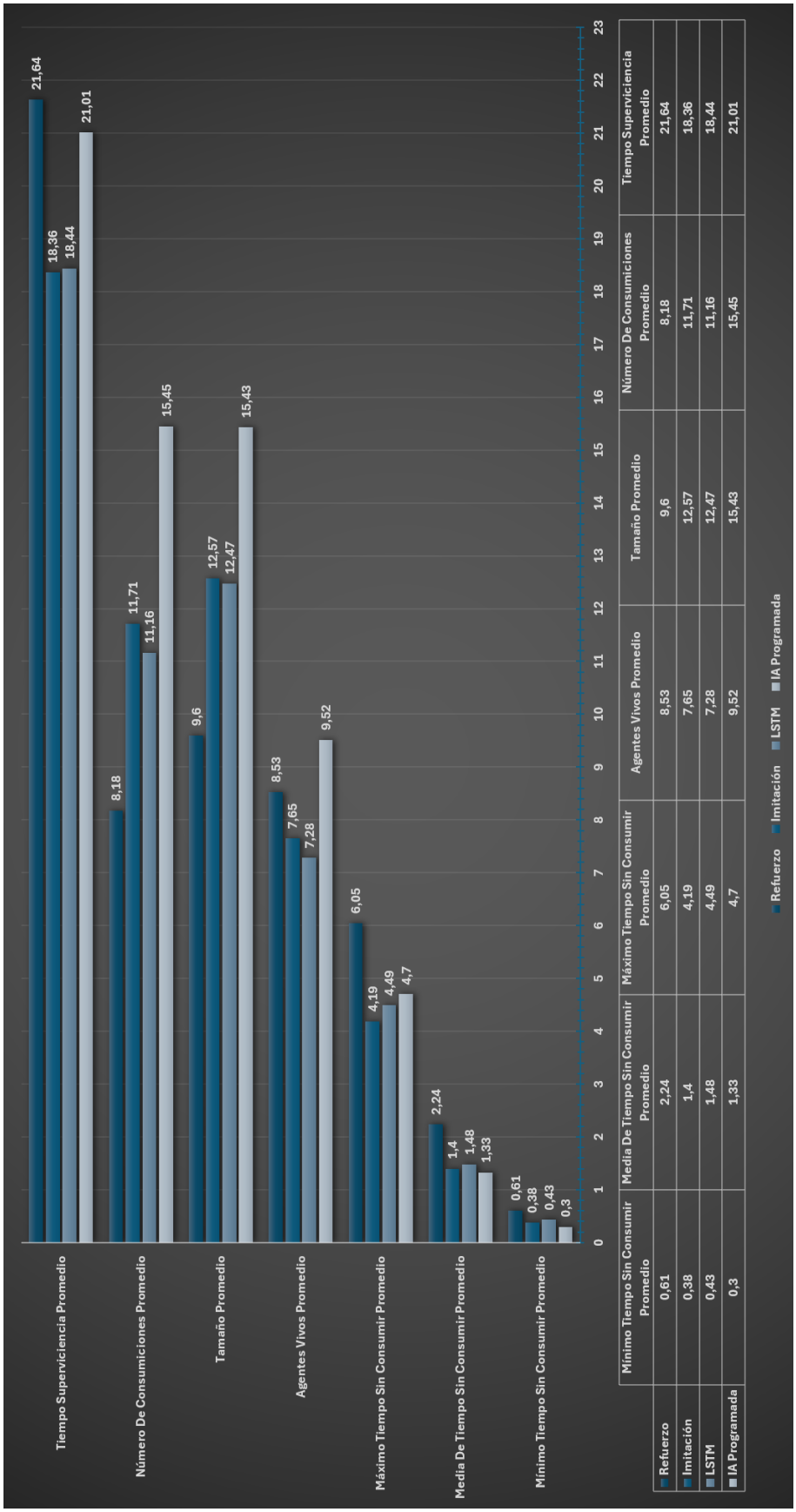


Figura 6.11: Gráfica de barras para comparar todos los tipos de entrenamiento

En cuanto a la valoración subjetiva, si se realiza una gráfica de barras que compare todos los datos se obtiene 6.11, permitiendo apreciar como, ignorando la anomalía en los campos de tiempo de supervivencia y agentes vivos ya explicada en la valoración subjetiva del entrenamiento por refuerzo, el entrenamiento por refuerzo se aleja bastante de la IA programada, con LSTM empieza a ser una mejor opción que se va aproximando a los resultados deseados y con imitación incluso mejora a este último por un poco de diferencia.

Finalmente, considerando todos los puntos mencionados para tomar unas valoraciones y remarcando que todas estos modelos se han de valorar teniendo en cuenta la importancia que tienen ciertos factores como el tipo de problema, su complejidad o con que recursos se cuente, para cada entrenamiento concluimos:

- El entrenamiento por refuerzo, como ya ha quedado claro, no ha obtenido buenos resultados. Para solucionarlo, entre diversas opciones, se podría optar por por aumentar la complejidad de la red o el tiempo de entrenamiento, pero en cualquier caso, si que ha sido útil para realizar pruebas y otras tareas que puedan requerir reiniciar el proceso.
- El entrenamiento con imitación es el que mejor ha funcionado con diferencia, pero tiene también bastantes inconvenientes. Dejando a un lado el esfuerzo extra, se puede considerar que en este método hay una dependencia entre los resultados y como de buenas son las demostraciones, de tal forma que ha mejores demostraciones, menos tiempo necesario y mejores resultados (tan bueno como el de las demostraciones). Es por ello que es recomendable únicamente si estos recursos están disponibles, tratándose de una muy buena opción si es el caso.
- Por último, el entrenamiento con LSTM se encontraría en un punto medio entre ambos anteriores, puesto que realmente no necesita ningún tipo de elemento adicional consiguiendo resultados más que admisibles. Sin embargo, el problema recae en la complejidad, puesto que en este tipo de entrenamiento se puede considerar que se entrenan dos redes simultáneamente: la propia red y la encargada de manejar la memoria. Es por ello que para este proyecto se redujo el número de nodos exclusivamente por este hecho y aun con esto es la que más tiempo ha requerido.

CAPÍTULO 7
Ampliaciones

En este apartado se tratará un aspecto que se ha programado, pero que realmente no hubiese sido necesario y que tampoco aporta nada demasiado útil para las conclusiones ya expuestas.

7.1 Añadir los resultados al *gameplay*

Para sacar provecho más práctico de las redes se ha realizado una última versión del juego que permita al usuario decidir el porcentaje de agentes que la usarán. Debido a los resultados, solamente se usará la entrenada con imitación, pues es la única que tiene un comportamiento suficientemente competitivo (la entrenada por LSTM era también una buena opción, pero aún tiene fallos que pueden afectar al *gameplay*).

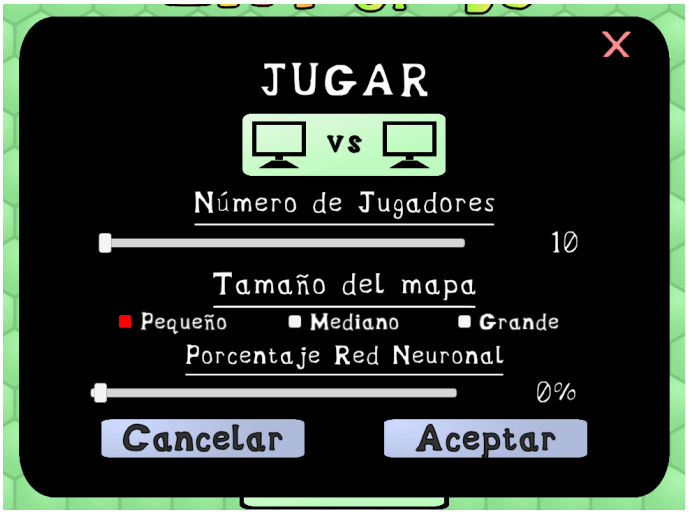


Figura 7.1: Nueva opción del porcentaje en el menú de jugar.

En la imagen 7.1 se pude observar la nueva opción añadida que permite al usuario interactuar con la nueva funcionalidad. En cuanto a programación, ha sido necesario comentar todas aquellas partes creadas para permitir mantener escenario infinitamente y el sistema de recompensas principalmente. Luego añadir una nueva entrada en *InformacionMapa* con el porcentaje, y modificar levemente la clase *ControladorAgentes* para que la tenga en cuenta a la hora de iniciar los agentes.

Con todo esto ya quedaría implementado, incorporando la posibilidad de encontrar agentes que se sientan algo más orgánicos y no tan robotizados como los originales[1].

[1] En https://github.com/FranNogui/EatOrBeEaten_Final se encuentra el proyecto de Git con esta versión.

CAPÍTULO 8
Conclusiones

Después de todo este proyecto a conseguido ser bastante útil para, principalmente, mejorar y perfeccionar el uso del motor de videojuegos Unity, para adquirir algo más de experiencia en el desarrollo de videojuegos, para ser capaz de evaluar alternativas de las que desarrollar una pequeña investigación obteniendo diversas conclusiones, y para adentrarse un poco más en el mundo de las redes neuronales y la inteligencia artificial en los videojuegos.

El primer desafío y objetivo cumplido ha sido el de programar y diseñar el videojuego desde cero, con el trabajo adicional de realizarlo manteniendo la idea de, en un futuro, ampliarlo con un nuevo sistema del que, al momento de concebir e implementar dicho videojuego, era desconocido su funcionamiento. Esto a llevando a que a la hora de llevar a cabo dicha ampliación se hayan requerido de adaptaciones e innovaciones con respecto a lo encontrado en la web. Sin embargo, se ha conseguido que todo marchará de una forma más que aceptable.

En un principio se había propuesto utilizar *ML-Agents* junto con otras tecnologías alternativas como material de estudio, no obstante *ML-Agents* ha acabado siendo la única opción que se adecue bastante a las expectativas, existiendo otras alternativas que, pese a ser también viables, impliquen tareas que se han considerado fuera del propósito de este proyecto. Así, se ha optado por utilizar el gran repertorio de herramientas que ofrece *ML-Agents*, usando finalmente: el sistema de aprendizaje por refuerzo, el algoritmo GAIL basado en la imitación y la incorporación de un sistema de memoria (apartado que ha implicado a su vez el uso de *Stacked vectors* y LSTM).

Con esto decidido, inicia el desarrollo de la nueva versión del videojuego que incorpore *ML-Agents*. En este momento se plantea la creación de una escena con partidas simultaneas para llevar a cabo los entrenamientos de la forma más eficiente posible. Tras programar todo lo necesario, da comienzo la fase de los entrenamientos, siendo una de las más largas si además se incluye todas las pruebas previas necesarias para alcanzar un estado estable y prometedor, además de, con todo esto completado, dar el tercer objetivo como cumplido.

Finalmente se obtienen los resultados de todos los entrenamientos, los cuales permiten obtener una conclusión de cada tipo de entrenamiento realizado, siendo estas:

- El entrenamiento por refuerzo no obtiene buenos resultados, aunque cabe la posibilidad de mejora aumentando la complejidad de la red y los tiempos de entrenamiento. Sin embargo, un punto positivo es que se trata de un método versátil por su rapidez a la hora de entrenar.
- Con imitación obtiene los mejores resultados, pero también es el que necesita de más esfuerzo y el que más depende de elementos extra.

- Con LSTM es una buena opción intermedia fácil de implementar una vez se tienen las bases y que ha proporcionado buenos resultados, pero debido a la complejidad por la naturaleza de su funcionamiento es el más lento a la hora de entrenar.

Como se puede observar, esto implica que el último objetivo también ha acabado siendo completado y que, por tanto, todas las metas planteadas al comienzo del desarrollo, a excepción de utilizar opciones fuera de *ML-Agents*, han sido alcanzados con resultados bastantes satisfactorios.

CAPÍTULO 9
Trabajo futuro

Para finalizar este trabajo se verán algunos puntos a expandir que no se han realizado pero que podrían ser interesantes para ampliar las conclusiones.

9.1 Mezcla y nuevas formas de entrenamiento

Una muy buena forma de ampliar el contenido del trabajo es probando más variedad de algoritmos o incluso mezclando los ya utilizados. Concretamente este trabajo se ha realizado usando *Proximal Policy Optimization* (PPO) siendo el que viene por defecto en las configuraciones de *ML-Agents*, existiendo una alternativa llamada *Soft Actor-Critic* (SAC). Con esto se podría llevar a cabo, incluyendo los ya hechos:

- Refuerzo en PPO / SAC.
- GAIL en PPO / SAC.
- *Stacked vectors* en PPO / SAC.
- LSTM en PPO / SAC.
- GAIL + *Stacked vectors* en PPO /SAC.
- GAIL + LSTM en PPO / SAC.
- *Stacked vectors* + LSTM en PPO / SAC.
- GAIL + *Stacked vectors* + LSTM en PPO / SAC.

Como se puede ver, este apartado tiene potencial para obtener más información sobre el funcionamiento de los entrenamientos y algoritmos.

9.2 Nuevos escenarios diferentes

Otra posibilidad es la de crear diferentes tipos de escenarios/mapas, de tal forma que los agentes deban adaptarse a más situaciones. Además de la nueva complejidad aparece un nuevo desafío, y es que la actual IA programada no funcionaría correctamente en estos escenarios, y por tanto habría dos opciones: programar una nueva para cada escenario o trabajar con las redes en base a la ausencia de esta IA (no podría usarse imitación salvo que se grabase manualmente).

9.3 Competición entre agentes

También podría aportar datos interesantes llevar a cabo una competición entre las distintas redes entrenadas. Para ello sería necesario crear una nueva versión del juego que sea capaz de registrar victorias, tiempo de supervivencia y cualquier otra información de utilidad de un conjunto de partidas simultáneas. Por otro lado, aunque podría parecer interesante la posibilidad de transformarlo en un juego por equipos, dado la naturaleza de la competición, esto implicaría también la necesidad de alterar el comportamiento de los agentes drásticamente. Esto último no significa que no se debería de hacer, pero sí que en caso de realizarse habría que tomarlo como un punto aparte donde se esta entrenando otra IA totalmente distinta.

9.4 Mejoras en el código y variantes

Finalmente se encuentran la opción de mejorar el actual estado del código, puesto que existen muchas zonas del mismo abiertas a mejoras o cambios. Específicamente el apartado de sistema de recompensas es uno de los más importantes y, a su vez, uno de los menos rígidos de todos los sistemas, siendo susceptible a multitud de cambios y ajustes que deciden como será el comportamiento del agente. Por otra parte, en cuanto a la posibilidad de cambiar elementos, existe una bastante interesante que puede ser el cambiar la salida de la red de continua a discreta, cambio del cual el entrenamiento LSTM puede aprovecharse puesto que en la documentación [5] se afirma que funciona mucho mejor en este tipo de escenarios. Por último, si se desea aumentar la complejidad del problema, se podrían desarrollar nuevas mecánicas como la posibilidad de realizar esquives, o añadir otros tipos de alimentos que fueran perjudiciales, entre muchas más posibilidades igualmente válidas.

Bibliografía

[1] Colaboradores de Wikipedia. Motor de videojuego [en línea]. Wikipedia, La enciclopedia libre, 2024 [fecha de consulta: 25 de enero del 2024]. Disponible en <https://es.wikipedia.org/w/index.php?title=Motor_de_videojuego&oldid=157681249>

[2] Documentación sobre el motor de Unreal Engine 5: https://docs.unrealengine.com/5.3/es-ES/

[3] Documentación sobre las herramientas de Unity 3D: https://docs.unity3d.com/Manual/index.html

[4] Página con información sobre los archivos de entrenamiento: https://unity-technologies.github.io/ml-agents/Training-Configuration-File/#self-play

[5] Página de Github con la información del funcionamiento básico de LSTM en ML-Agents: https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Feature-Memory.md

[6] Página de Github con información sobre el funcionamiento del *Self-play*: https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Training-Self-Play.md

[7] Página de Github con información sobre los agentes de ML-Agents: https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Learning-Environment-Design-Agents.md

[8] Página de GitHub con información sobre los motores de videojuegos, creado por el usuario *raysan5*: https://gist.github.com/raysan5/909dc6cf33ed40223eb0dfe625c0de74

[9] Página de GitHub con instrucciones de como instalar ML-Agents: https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Installation.md

[10] Página de GitHub de ML-Agents: https://github.com/Unity-Technologies/ml-agents/tree/develop

[11] Shao, K., Tang, Z., Zhu, Y., Li, N., & Zhao, D. (2019). A Survey of Deep Reinforcement Learning in Video Games.

[12] Song, J., Ren, H., Sadigh, D., & Ermon, S. (2018). Multi-Agent Generative Adversarial Imitation Learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), Advances in Neural Information Processing Systems (Vol. 31). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/240c945bb72980130446fc2b40fbb8e0-Paper.pdf

[13] Torrado, R. R., Bontrager, P., Togelius, J., Liu, J., & Perez-Liebana, D. (2018). Deep Reinforcement Learning for General Video Game AI. 2018 IEEE Conference on Computational Intelligence and Games (CIG), 1–8. https://doi.org/10.1109/CIG.2018.8490422

[14] Wan, S., & Kaneko, T. (2017). Imitation Learning for Playing Shogi Based on Generative Adversarial Networks. 2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI), 92–95. https://doi.org/10.1109/TAAI.2017.17

[15] Zheng, Y. (2019). Reinforcement Learning and Video Games.

APÉNDICE A
Relación de clases en el videojuego

En la figura A.1 se puede observar una casi perfecta relación entre todas las clases que conforman el proyecto. Además, se han añadido bloques que indican dentro de que entorno se encuentran dichas clases. Así, aquí se tratarán algunos puntos que enriquecen y marcan mejor el significado del diagrama:

- Las clases de *ControladorMenuSonidos.cs*, *SonidoBotones.cs* y *ControladorMusica.cs* son utilizadas en ambas escenas, y son independientes del resto de clases. Así, se añaden directamente en los componentes de *Unity* que las necesiten y funcionan por cuenta propia.
- *ControladorMenuPrincipal.cs* es la clase que se encarga de controlar por completo las escena del menú principal, asegurando el correcto funcionamiento de los menús y sus componentes, y siendo responsable de modificar el *Scriptable Object* de clase *InformacionMapa.cs* que permite transferir información entre escenas.
- *ControladorMapa.cs* es la case encargada de iniciar todo lo relacionado con el inicio de la partida, ajustando así las dos secciones principales de *ControladorAgentes.cs* y *AlimentoSpawner.cs* según la información del *Scriptable Object*.
- *ControladorAgentes.cs* es una de las piezas clave que dirigen la partida, encargándose de, como su propio nombre indica, controlar toda la creación y mantenimiento de los agentes, de intermediario entre estos y la cámara para realizar tareas como que esta siga a un agente, y de mostrar pantallas al acabar la partida a través de *ControladorMenuOpciones.cs*.
- Dentro del agente básicamente contamos con *ControladorAgente.cs* como intermediario entre un agente y *ControladorAgentes.cs* (es decir, entre el agente y el exterior), siendo también la que dirige la mayoría de módulos que conforman un agente. Además, se encuentra la excepción del *ControladorLayer.cs* que se comunica directamente con *ControladorTamanyo.cs* para obtener la información que necesita. *ObjetoDeRayCast.cs* es utilizado por 3 clases para encapsular la información de los *raycasts*, que además fluye entre ellas. Finalmente, hay que entender *IAProgramada.cs* como una clase de la que *ModuloMovimiento.cs* obtiene información para sus cálculos.
- La sección de *AlimentoSpawner.cs* sigue el mismo patrón que el de *ControladorAgentes.cs*, sirviendo de intermediario entre los alimentos a través de *ControladorAlimento.cs*, cuya única tarea dividida es la de manejar los sonidos con *AlimentoSonidos.cs*.

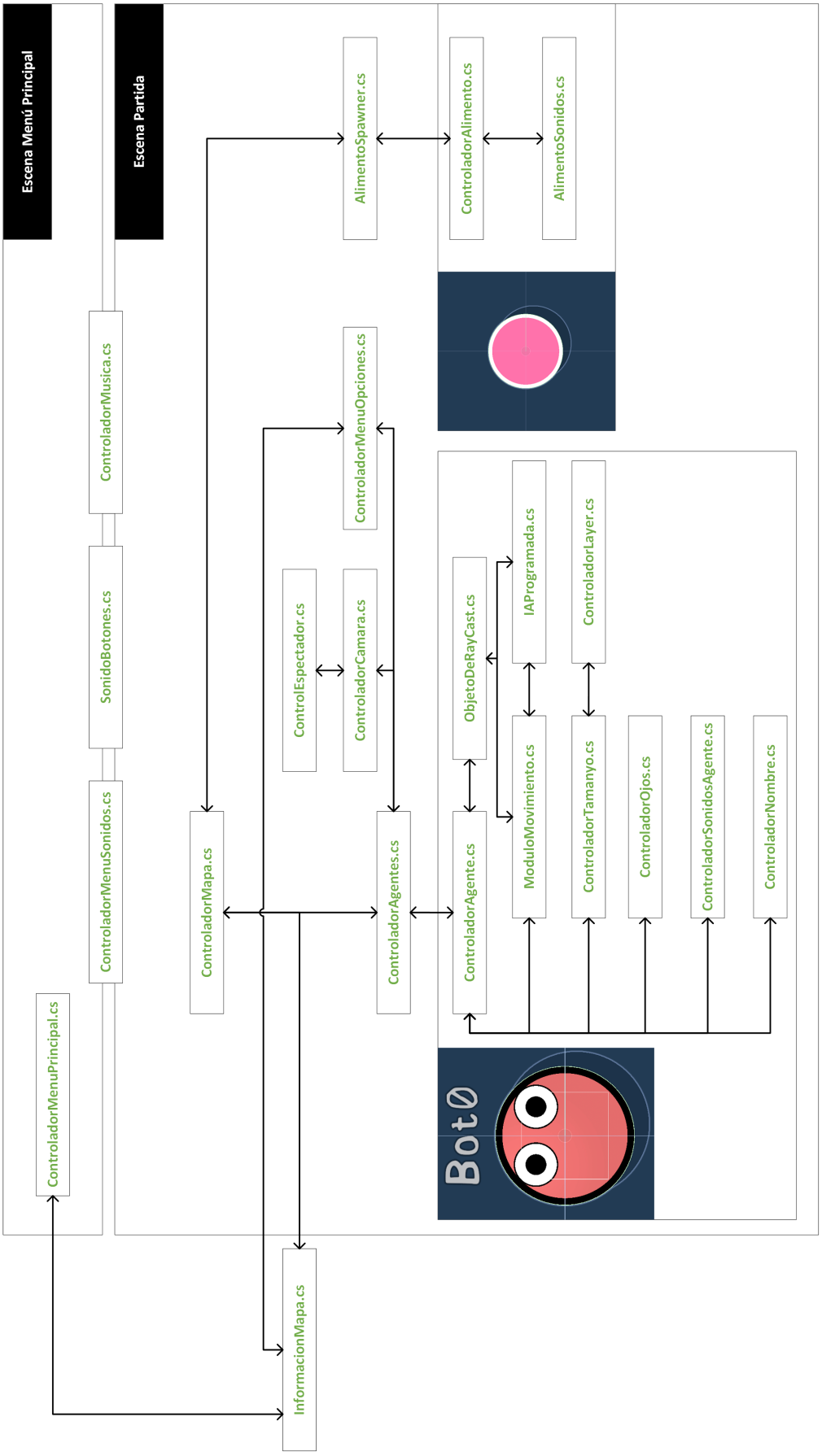


Figura A.1: Relación entre todas las clases en el videojuego original.

APÉNDICE B
Código del videojuego

En este apéndice se encuentra los enlaces a todo el código que compone el videojuego desarrollado antes de incorporar elementos relacionados con las redes neuronales:

1. Código relacionado con los agentes.
    a) ControladorAgente.cs
    b) ControladorAgentes.cs
    c) ControladorLayer.cs
    d) ControladorNombre.cs
    e) ControladorOjos.cs
    f) ControladorSonidosAgente.cs
    g) ControladorTamanyo.cs
    h) IAProgramada.cs
    i) ModuloMovimiento.cs
    j) ObjetoDeRayCast.cs
2. Código relacionada con los alimentos.
    a) AlimentoSonidos.cs
    b) AlimentoSpawner.cs
    c) ControladorAlimento.cs
3. Código relacionado con los botónes.
    a) SonidoBotones.cs
4. Código de control de la cámara.
    a) ControladorCamara.cs
5. Código relacionado con las interfaces.
    a) ControladorMenuOpciones.cs
    b) ControladorMenuPrincipal.cs
    c) MenuSonidos.cs
6. Código de control del jugador.
    a) ControlEspectador.cs

7. Código relacionado con el mapa.
    a) ControladorMapa.cs
    b) InformacionMapa.cs
8. Código de control de la música.
    a) ControladorMusica.cs

APÉNDICE C
Código del videojuego para el entrenamiento

En este segundo apéndice contamos con el código que se encuentra en el juego alterado para realizar los entrenamientos necesarios, así como directorios importantes. Aunque muchas clases han sido inalteradas, hay también algunas que si han sufrido bastantes cambios:

1. Código relacionado con los agentes.
    a) ControladorAgente.cs
    b) ControladorAgentes.cs
    c) ControladorLayer.cs
    d) ControladorNombre.cs
    e) ControladorOjos.cs
    f) ControladorSonidosAgente.cs
    g) ControladorTamanyo.cs
    h) IAProgramada.cs
    i) IARedNeuronal.cs
    j) ModuloMovimiento.cs
    k) ObjetoDeRayCast.cs
2. Código relacionada con los alimentos.
    a) AlimentoSonidos.cs
    b) AlimentoSpawner.cs
    c) ControladorAlimento.cs
3. Código relacionado con los botónes.
    a) SonidoBotones.cs
4. Código de control de la cámara.
    a) ControladorCamara.cs
5. Código relacionado con las interfaces.
    a) ControladorMenuOpciones.cs

   b) ControladorMenuPrincipal.cs
   c) MenuSonidos.cs

6. Código de control del jugador.
   a) ControlEspectador.cs
7. Código relacionado con el mapa.
   a) ControladorMapa.cs
   b) InformacionMapa.cs
8. Código de control de la música.
   a) ControladorMusica.cs
9. Directorios importantes:
   a) Configuraciones de las redes neuronales
   b) Directorio con las demos grabadas
   c) Directorio con los resultados de los entrenamientos

APÉNDICE D
Game Design Document

A continuación se encuentra el Game Design Document con toda la información necesaria para entender el concepto y la idea detrás del videojuego.

EAT OR BE EATEN

Game Design Document

AUTOR:
FRANCISCO JOSÉ NOGUERA GUIJARRO

Contenido

Tabla de ilustraciones

1. Introducción

1.1. Título del juego

Eat or be Eaten

1.2. Concepto del juego

Compite contra otros seres como tú por sobrevivir en un entorno hostil. Para ello, aliméntate de los elementos del ambiente o de los enemigos para aumentar tu tamaño y asegurar tu supervivencia.

2. General

Género: Juego de acción

Plataforma: PC

Público Objetivo: Apto para cualquier tipo de jugador por su simplicidad y facilidad

PEGI: +7 debido al mínimo nivel de violencia

Características:

Se trata de un juego simple pero frenético por situar al jugador en una constante competición por sobrevivir donde, a no ser que hayas conseguido avanzar bastante como para ser el de mayor tamaño de la partida, estarás siempre en peligro de ser eliminado por otros entes.

3. Gameplay

3.1. Modos de juego

Existirán dos modos de juego principales: máquina contra máquina y jugador contra máquina.

Con esto se pretende que exista tanto la posibilidad de que el jugador interactué en el desarrollo de una partida, y que también pueda simplemente actuar como un espectador y verlo como si se tratase de una exhibición. Sin embargo, esta diferencia solo cambiará los controles del jugador y cuáles serán sus posibles acciones durante una partida.

3.2. Movimiento y acciones del jugador

Para definir de mejor forma que es lo que podrá realizar el jugador tendremos en cuenta en que modo de juego nos encontramos:

Máquina contra máquina:

En este modo el jugador podrá mover la cámara libremente por todo el mapa, o seguir a alguno de los NPCs que se encuentran en la partida. En cualquier momento se puede abrir el menú de opciones para abandonar la partida o cambiar los ajustes.

Ilustración 1. Botón para acelerar la partida

Además, con el botón de la *Ilustración 1* se puede acelerar la partida, cambiando la velocidad de esta entre los valores 1.0, 1.5, 2.0, 2.5 y 3.0 progresivamente.

Jugador contra máquina:

En este otro modo, el jugador tomará control de uno de los entes. Así, podrá moverlo por el mapa alimentándose de los elementos del entorno con el fin de aumentar su tamaño para posteriormente eliminar otros seres. De igual forma al otro modo, también se puede abrir el menú de opciones abandonar la partida o cambiar los ajustes.

Más detalles de los controles de ambos modos en el apartado 9.2. Controles.

4. Niveles de juego

Habrá un solo nivel en la práctica, ya que se tratará de un plano donde ocurrirá la partida, limitado por los cuatro lados.

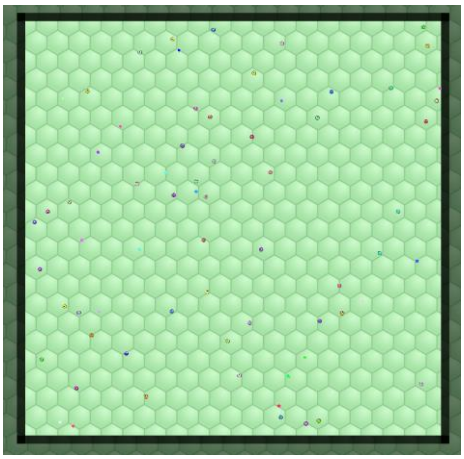


Ilustración 2. Ejemplo de nivel completo

Sin embargo, podrá cambiar de tamaño según decida el jugador, así como de cantidad de entes iniciales. De esta forma se puede crear cierta diversidad de posibilidades que requieren de alguna manera diferentes estilos de juego.

En concreto, el tamaño podrá ser:

- Pequeño
- Mediano
- Grande

, y la cantidad de entes entre 10 y 50 inclusive.

Ilustración 3. Opciones del mapa

En la figura superior se puede observar como se le permite al jugador decidir los campos ya mencionados justo antes de comenzar una partida. Así, por ejemplo, si se elige un mapa pequeño con la máxima cantidad de jugadores (50) será una partida rápida e intensa al comienzo y rápida al final, mientras que si se elige el mapa grande con menos jugadores será algo rápida al comienzo, pero muy lenta al final.

5. Personajes

Ilustración 4. Imágenes de los distintos agentes

Existen tres posibles agentes que pueden aparecer en una partida. Sin embargo, las diferencias entre ellos son simplemente visuales, manteniendo un comportamiento funcional idéntico. Además, cada agente puede variar de color teniendo en cuenta que, con el fin de mantener una apariencia visual agradable y coherente, dicho color deber tener al menos una de sus componentes RGB máxima.

Por otro lado, también con fines sobre todo estéticos, los ojos de los entes siguen ciertos comportamientos que sirven hasta cierto punto como mecánica para el jugador, pero principalmente para dotar de vida al juego. En concreto, estos marcan la dirección hacia la que se mueve un agente en un momento dado, y de forma aleatoria realizan una animación de parpadeo.

Finalmente, como se puede ver en la *Ilustración 2*, cuentan con una etiqueta con un nombre que les permite diferenciarse de cualquier otro agente, y en cuanto a la eliminación entre agentes, visualmente se representa por una animación donde el ente eliminado se encoje y mueve hacia el centro del otro de forma suave y progresiva, manteniendo cierta inercia con respecto al movimiento del agente eliminador.

6. Inteligencia de los agentes

Pese a uno estar programado como tal, el comportamiento de los agentes se puede representar mediante una máquina de estados finito:

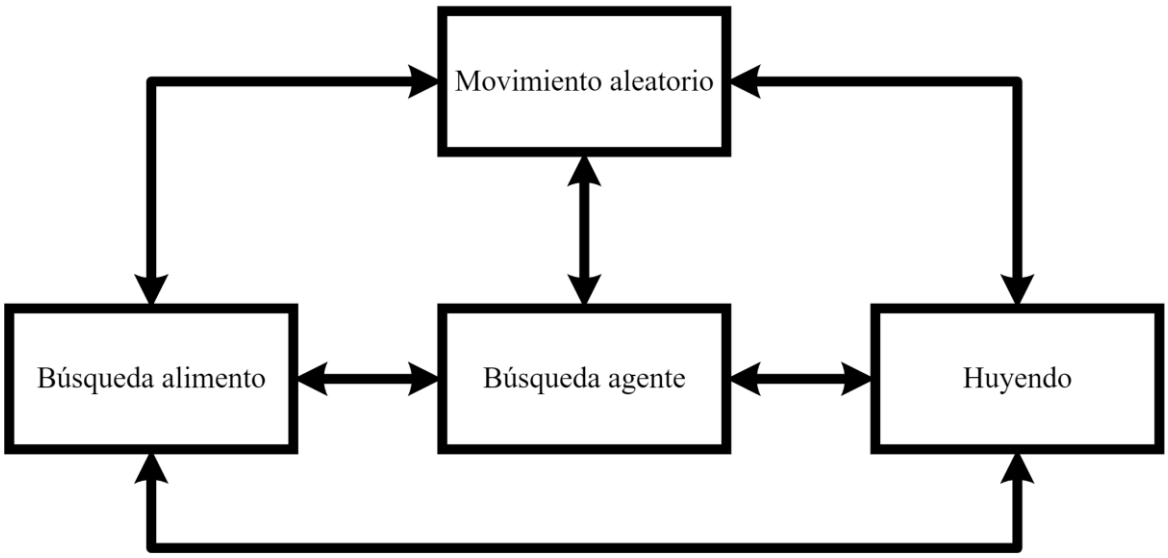


Ilustración 5. Máquina de estado de los agentes

Como se puede ver en la figura superior, un agente se puede encontrar en cuatro estados distintos y puede viajar de uno dado a cualquier otro, creando un grafo totalmente conexo. Esto se debe a que la decisión de realizar una cosa u otra depende de lo que observa el agente de su entorno a través de una multitud de *RayCasts*. En concreto cada estado funciona de la siguiente forma:

- Movimiento aleatorio: El agente se moverá de forma aleatoria cuando no perciba nada de interés en su entorno (ni alimentos ni agentes).

- Búsqueda alimento: Si el agente observa uno o varios alimentos en sus inmediaciones, decidirá cuál es más conveniente según tamaño y distancia, para posteriormente moverse en su dirección.

- Búsqueda agente: Siguiendo la misma lógica a búsqueda de alimento, si el agente percibe otros agentes en inferioridad, decidirá cuál es mejor y lo perseguirá.

- Huyendo: En caso de observar agentes de mayor tamaño, decidirá cuál es más peligroso según distancia y se moverá en la dirección opuesta a dicho agente. En caso de encontrar paredes, se realizará un movimiento que permita esquivar la pared en cierta medida, y es posible que el agente decida cambiar a *Búsqueda alimento* o *Búsqueda agente* si es conveniente y no interfiere en gran medida en la huida.

7. Elementos del juego

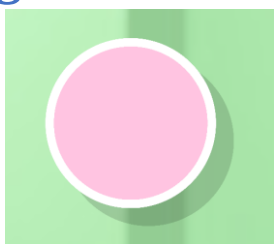


Ilustración 6. Alimento en una partida

A parte de los propios entes, en una partida se pueden encontrar alimentos que sirven para aumentar de tamaño siempre, de tal forma que los agentes no se queden estancados en caso de ir muy atrás en la partida o para simplemente conseguir ventaja. Estos alimentos siguen el mismo esquema de color que los agentes, aunque a diferencia de estos, tienen un borde blanco en lugar de negro.

Son estáticos y tiene un volumen que puede variar entre 0.5 y 1, haciendo que al principio permitan aumentar relativamente rápido de tamaño, pero según avanza la partida sea necesario consumir muchos más para que influyan notablemente. Para compensar esto, la ratio de aparición de estos variará progresivamente a lo largo de la partida. Así, al inicio aparecerán muchos menos que al final, momento en el que es posible incluso que se encuentren en partida la máxima cantidad de los mimos.

8. Audio del juego

El audio se puede dividir en dos secciones principales: música y efectos SFX.

Música:

En total se usan cuatro pistas musicales en todo el juego, una que se reproduce constantemente en el menú principal y tres que se alternan de forma aleatoria durante el desarrollo de una partida.

Efectos SFX:

En cuanto a efectos, en tema de interfaces existen sonidos al cerrar y abrir cajas de menús, así como al colocar el ratón sobre alguno de los botones. Finalmente, se reproducirán "*pops*" aleatorios cuando un ente elimine un alimento o a otro agente.

9. Interacción

9.1. Interfaz de juego

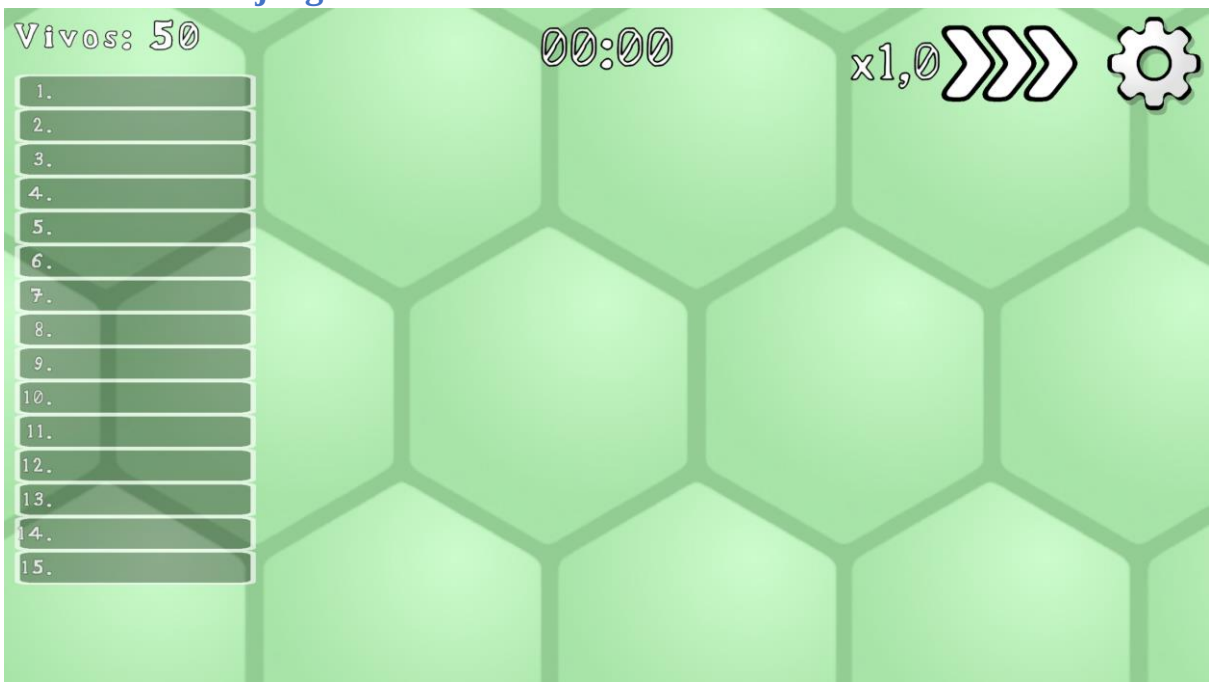


Ilustración 7. Elementos de la interfaz en partida

Aparte de los distintos menús que se pueden ver en las pantallas del apartado 9.3. Jerarquía de pantallas, contamos con una interfaz en partida que se compone de 5 elementos principales:

1. **Contador de agentes vivos:** Se trata del elemento superior izquierdo y, como su nombre indica, mantiene constancia del número actual de agentes no eliminados.

2. **Lista TOP:** Debajo de la cantidad de agentes vivos se encuentra una lista que muestra en orden los 15 agentes con mejor puntuación de la partida (la puntuación corresponde con el tamaño del agente). Además, cuando alguno ha sido eliminado, aparecerá su Nick en rojo.

3. **Contador del tiempo de partida:** El contador que se encuentra en el centro arriba de la pantalla muestra el tiempo total de partida.

4. **Botón para aumentar velocidad de partida:** El botón de tres flechas a la derecha del contador del tiempo varia la velocidad entre los valores 1.0, 1.5, 2.0, 2.5 y 3.0 progresivamente. Este cambio también afecta a la velocidad del contador del tiempo.

5. **Engranaje para abrir las opciones:** Botón que se encuentra en la parte superior derecha, que permite abrir el menú de opciones como alternativa al botón de escape del teclado.

9.2. Controles

Para indicar los controles diferenciaremos aquellos que funcionan globalmente en ambos modos de juego, especificando para que sirven en cada uno, y aquellos que son exclusivos cuando se trata de máquina contra máquina:

Globales:

Tecla / Botón	Utilidad
W A S D	Movimiento del personaje o espectador
Q	Ocultar/Mostrar Nick de los agentes
Tabulador	Ocultar/Mostrar lista TOP
Escape o Botón Engranaje	Abrir/Cerrar menú de opciones en partida

Exclusivo de máquina contra máquina:

Tecla / Botón	Utilidad
Rueda del ratón	Controlar el zoom de la cámara
Shift	Aumenta la velocidad
Click en agente	Seguir con la cámara al agente seleccionado
Click en Nick de la lista TOP	Seguir con la cámara al agente seleccionado
Click en el botón de tres flechas	Aumentar velocidad de la partida

9.3. Jerarquía de pantallas

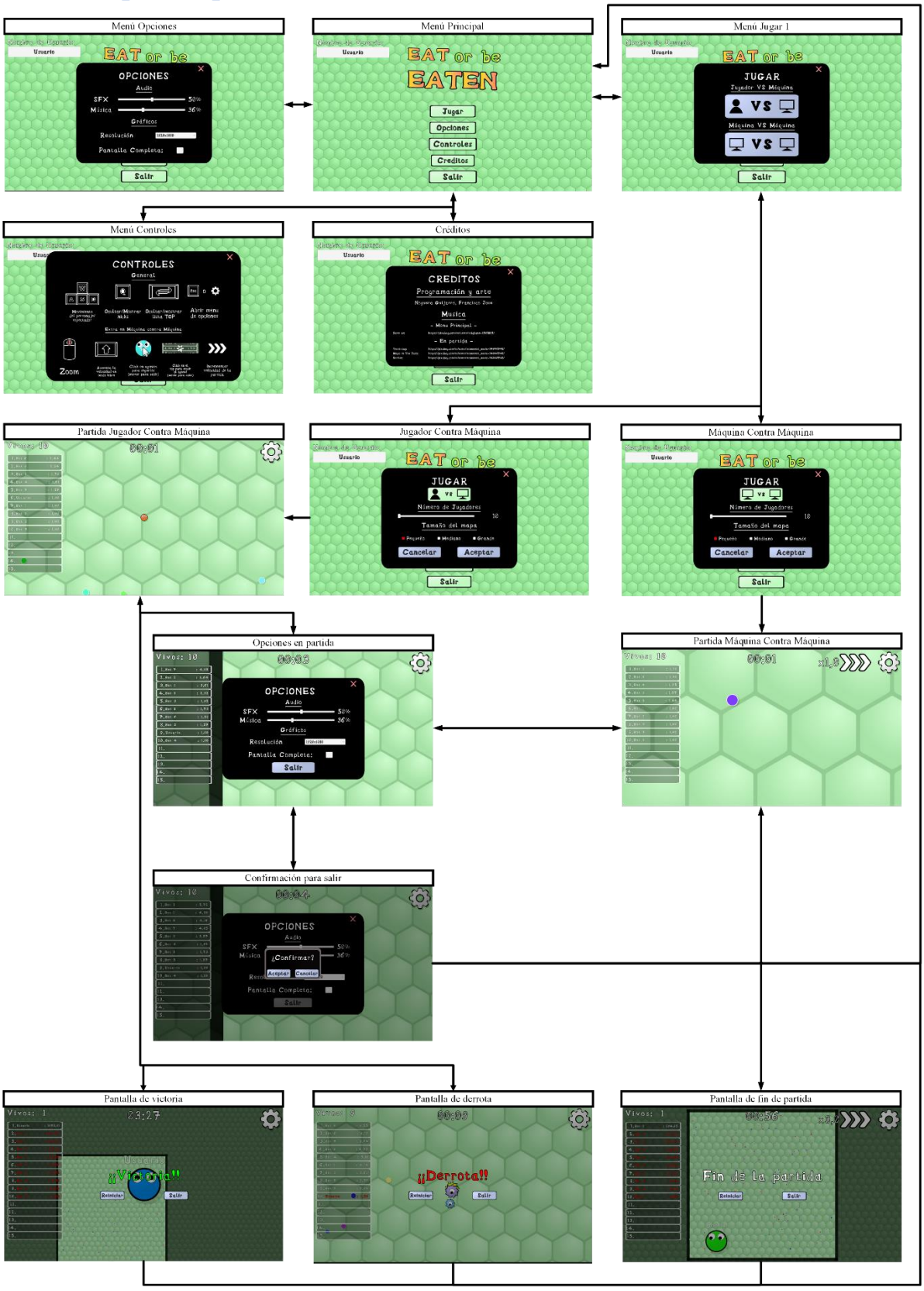


Ilustración 8. Esquema con la jerarquía de pantallas

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**		X		
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**				X
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Tras valorar todos los objetivos propuestos y realizar una pequeña tarea de análisis del trabajo realizado, se puede concluir que ni el contenido ni los resultados guardan relación directa con ninguno de estos. Aun así, pese a no existir esta relación directa entre el proyecto y algún apartado de los ODS, gracias a haber tenido que trabajar directamente en el proceso de entrenamiento de una red neuronal me ha permitido vivir de primera mano la cantidad de tiempo y recursos, más concretamente energéticos, que esto requiere.

Es evidente que la envergadura de este trabajo es muy limitada al tratarse de algo particular y realizado por una solo persona, lo que implica que el impacto global es mínimo. Sin embargo, desde un punto de vista "local" sí que se ha podido apreciar cambios considerables en cuanto al tiempo que pasa el sistema trabajando con respecto al uso cotidiano del mismo. Es por estos hechos que la reflexión sobre los ODS se va a enfocar en analizar cuáles han sido con más detalle los cambios personales en cuanto a consumo energético, para posteriormente realizar una búsqueda e informarse sobre este tema, así de cómo se aborda cuando se trata de equipos y proyectos más grandes.

Comenzando por los cambios experimentados, el entrenamiento de las redes ha sido el proceso más largo y costoso de todo el trabajo, y es que es importante tener en cuenta que en muchas ocasiones mantener el sistema trabajando en esta tarea limita al usuario en cuanto al uso que este puede dar del mismo. Así, ha habido situaciones en las que, al no respetar debidamente estas restricciones de uso, el proceso de entrenamiento ha finalizado abruptamente con un error. Es por todo esto que muchas veces ha sido necesario mantener el sistema activo en horarios nocturnos exclusivamente trabajando en estas tareas, circunstancia que nunca se había dado antes. Claramente no ha supuesto grandes problemas a nivel personal, pero ha sido suficiente como para que por lo menos me haya llamado la atención, decidiendo realizar esta reflexión en torno a este asunto.

Continuando con, ahora sí, la misma situación a gran escala, en el artículo https://www.theverge.com/24066646/ai-electricity-energy-watts-generative-consumption se realiza un análisis muy interesante que toca varios puntos que son importantes a considerar. Lo primero es que, con el paso del tiempo, las empresas han ido decidiendo mantener en secreto cierta información que complica conocer exactamente cual es la situación real de todo esto. Sin embargo, también se abre la posibilidad de que, por al menos razones económicas, las empresas también tomen medidas para reducir el consumo, adjuntando el estudio https://arxiv.org/pdf/2104.10350 que desarrolla en detalle muchos resultados con posibles soluciones a este problema.

Por otro lado, siguiendo el mismo artículo, también se plantean algunos conceptos que son interesantes para entender mejor la implicación de las IAs en el consumo energético. Lo primero es la distinción entre la fase de entrenamiento, la cual para los modelos actuales como GPT-3 se hacen estimaciones que llevan a consumos muy elevados (según el artículo, y para este ejemplo, equivaldrían al consumo anual de 130 casas en Estados Unidos), y la fase de mantenimiento y uso de los resultados obtenidos. Para esta segunda fase también se cuenta con el estudio en https://arxiv.org/pdf/2311.16863 que permite obtener conclusiones muy variadas donde se distinguen todo tipo de salidas (una de estas conclusiones es que los modelos para generar imágenes consumen bastante más que aquellos de texto). Todos estos estudios y análisis otorgan las bases para deducir y prevenir sobre el futuro de los modelos en este ámbito.

Las ideas finales que trata de transmitir el artículo, y que considero pueden ser significativas y relevantes, son las de que en un futuro próximo sea necesario tomar medidas que, hasta cierto punto, incentiven a las empresas a ver este asunto como algo a valorar, y que de esta forma sí tomen soluciones que al final beneficiarán a todos.