Unity HEX BattleGame

Reinforcement Learning con Unity ML-Agents para entornos Multi-Agente y toma de decisiones bajo demanda

Reinforcement Learning with Unity ML-Agents for Multi-Agent Environments and On-Demand Decision Making

Fernando González Petit

Trabajo de Fin de Grado
Universidad de La Laguna
Septiembre 2020

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

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

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

Que la presente memoria titulada:

" *Unity HEX BattleGame: Reinforcement Learning con Unity ML-Agents para entornos Multi-Agente y toma de decisiones bajo demanda*"

ha sido realizada bajo su dirección por D. **Fernando González Petit**, con N.I.F. 54.059.959-T.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11de septiembre de2020

Agradecimientos

Especiales agradecimientos al tutor del TFG, José Demetrio Piñeiro Vera, y al cotutor, Jesús Miguel Torres Jorge, por dedicarme su tiempo aún durante estos tiempos de pandemia y por sus valiosas recomendaciones y feedback.

También a la comunidad de GameFaqs de Final Fantasy X por su excelente trabajo de ingeniería inversa sobre las mecánicas del sistema de combate del juego, y a mi amigo Eduardo, que un buen día me invitó a jugar a Star Wars: Imperial Assault, juego al que HEX le debe su culminación.

Por último, pero no menos importante (de hecho, todo lo contrario), a mis padres y a mi hermana por aguantarme y apoyarme a lo largo de mi carrera universitaria y de mi día a día.

Resumen

Este proyecto pretende ser pionero en la utilización de agentes entrenados con Aprendizaje por Refuerzo como parte de la IA de videojuegos. Más concretamente, su meta es estudiar su viabilidad y desempeño en Unity HEX BattleGame, diseñado por el autor con un *sistema de batalla táctico por turnos* donde tanto el jugador como los NPC se baten en duelo por la supervivencia de su especie. Este trabajo incluye los resultados del entrenamiento de agentes como dichos NPC para este tipo de entornos, usando técnicas modernas de Curriculum Learning y el *framework* de Unity ML Agents para adquirir los mejores resultados posibles con *toma de decisiones bajo demanda* en *entornos multi-agente*.

Los resultados observados exponen estrategias prometedoras para escenarios simples, con un buen ritmo de aprendizaje en los agentes, que además han desarrollado tendencias que no solo están a la par con su contrapartida heurística, sino que ocasionalmente demuestran ser más orgánicas. Técnicas de Curriculum Learning también han demostrado ser útiles para el entrenamiento de tareas más complicadas (como por ejemplo, el *movimiento* y la *adaptabilidad al terreno*), y por tanto, sería beneficioso abordar diferentes maneras de mejorar el comportamiento de los agentes, para el futuro de este y de otros juegos similares.

Palabras Clave: Aprendizaje por Refuerzo, Machine Learning, Unity MLAgents, Combate por Turnos, HEX

Abstract

This project pretends to be a step forward towards the utilization of agents trained with Reinforcement Learning on real-life videogame AI scenarios. Specifically, it aims to study their viability and performance in the Unity HEX BattleGame, designed by the author as a *turn-based, tactical, battle system* where both player and NPCs clash and fight for the survival of their species.
The study includes the results of training agents as the NPCs for said environments, using state of the art techniques from Curriculum Learning and the Unity ML Agents framework to accquire the best possible results with *on-demand decision making in multi-agent environments*.

Results show promising behaviors in simple scenarios, with agents learning steadily across the board and developing tendencies not only equal to their heuristic counterpart, but ocassionally more organic and measured. Curriculum Learning has also shown to be useful for the training of more difficult tasks (namely *movement* and *terrain adaptability*), and thus, it would be beneficial to expand on different approaches to further improve the agents' performance for the future of this and other similar games.

Keywords: Reinforcement Learning, Machine Learning, Unity MLAgents, Turn-Based Battle, HEX

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En esta primera sección se pretende introducir breve y concisamente el **Proyecto/Trabajo de Fin de Grado** desarrollado por *Fernando González Petit* para el *Grado en Ingeniería Informática* de la *Universidad de La Laguna* [1].
El proyecto presentado consiste en la implementación de técnicas de *Reinforcement Learning* en un videojuego de batalla por turnos, también diseñado en su totalidad por el autor.

1.1. Motivaciones Principales del Proyecto

Históricamente, los videojuegos de rol (RPG) han limitado la inteligencia de los enemigos a algoritmos deterministas o semi-deterministas. Aunque estos métodos suelen dar buenos resultados y garantizan comportarse de acuerdo a un diseño previo, también pueden carecer de cierto componente de flexibilidad y complejidad.

En este proyecto, se pretende utilizar técnicas de Deep Learning y aprendizaje por refuerzo para entenar el comportamiento de estos agentes, estudiando su viabilidad y factibilidad en escenarios de menor a mayor complejidad y anotando los progresos por fase. El diseño previo del entorno y del videojuego en sí se cubrirá también en detalle, poniendo especial atención en los componentes relevantes para el entrenamiento de los agentes.

Otro factor clave será el estudio de los componentes sociales del entorno multi-agente: los roles de los diferentes enemigos en el ecosistema, su grado de interacción con el entorno en correspondencia con el diseño previo y las diferencias encontradas con respecto a comportamientos heurísticos.

1.2. Antecedentes y Estado del Arte

Tradicionalmente, el RPG clásico ha optado por enfoques puramente deterministas en la codificación del comportamiento de los NPC. Sin embargo, esto no es la norma para todos los sub-géneros del videojuego.
En la práctica, el uso de Deep Learning es un **tema de investigación y actualidad**: desde agentes que desarrollen tareas in-game hasta los que toman el papel del jugador para completar niveles y tareas automáticamente. Este proyecto se encuadra en la primera categoría.

Desde proyectos independientes como *OpenAI* [3] y *Unity MLA-gents*, se plantean escenarios diversos en los que se entrenan agentes para conseguir comportamientos fácilmente aplicables a nuestro tipo de entornos (es, de hecho, la herramienta que proporciona este último la que ha sido utilizada para el desarrollo del TFG).
En la documentación oficial (en la **página de GitHub [4] del kit de MLAgents**), se proporcionan varios ejemplos sencillos que ilustran buenas prácticas en el diseño de agentes para escenarios de juego. Sin embargo, ninguno de estos presenta exactamente las mismas características que el proyecto a desarrollar –un entorno **multi-agente** en el que se **toman decisiones por turnos** (toma de decisiones bajo demanda)– por lo que se espera que acabe siendo una buena referencia o punto de partida para trabajos futuros en entornos similares.

1.3. Herramientas Utilizadas y Control de Versiones

A continuación, quedan enumeradas las herramientas utilizadas para el desarrollo del videojuego y el entrenamiento de los agentes:

- **Unity (Editor):** v2019.4.3f1
- **Unity ML Agents Kit:** Última versión estable [Release 6] (24/08/2020). Versiones de los paquetes adjuntos:
  - *com.unity.ml-agents (C#):* v1.3.0
  - *ml-agents (Python):* v0.19.0
  - *ml-agents-envs (Python):* v0.19.0
  - *gym-unity (Python):* v0.19.0
  - *Communicator (C#/Python):* v1.0.0
- **Anaconda (Python Env):** v4.8.4
- **Python:** v3.7

1.4. Objetivos

El objetivo general de este proyecto es aplicar técnicas de *Deep Learning y Aprendizaje por Refuerzo* para estudiar la factibilidad y viabilidad de la aplicación directa de los comportamientos obtenidos en un escenario de juego real.

En esta línea, se realizarán diversas rutinas de entrenamiento para una serie de agentes, usando también estrategias de *Curriculum Learning* para obtener resultados comparables con enfoques heurísticos previamente probados.

1.5. Línea de Trabajo y Fases del Desarrollo

En esta memoria se expondrán las fases principales del proyecto, indagando en tecnicismos, peculiaridades y dificultades presentadas cuando fuese necesario y prestando especial atención al diseño del videojuego propuesto y al papel de los agentes en el mismo.

El comportamiento de los agentes, los parámetros de los diferentes entrenamientos y sus progresos serán cubiertos en su propia sección. En cada iteración de los entrenamientos, se harán pruebas con diferentes parámetros –como la **función de recompensa** del agente o **características de la red neuronal** subyacente– para obtener los mejores resultados posibles, garantizar un progreso estable y un comportamiento acorde a los diseños propuestos.

1.6. Resultados Esperados

Con este proyecto se pretende obtener resultados constatables que prueben poder utilizar técnicas de Deep Learning para el modelado de NPC en videojuegos de estrategia por turnos, como es el caso de **HEX.**

En concreto, esto implica:

- Diseñar agentes independientes y con metas significativas pero sencillas para el videojuego.
- Preparar y utilizar entornos para el entrenamiento de estos agentes, analizando los resultados obtenidos y la efectividad del método frente a la heurística.
- Diseñar agentes con espacios de acción más amplios y medir su desempeño con respecto a las siguientes fases.
- Aplicar técnicas de Curriculum Learning para mejorar los resultados y su adaptabilidad al tablero de juego y a nuevas habilidades/unidades.
- Contrastar y comparar los datos estadísticos de los entrenamientos, buscando posibles mejoras y optimizaciones.

2. Documento de Game Design: HEX

HEX pretende ser un videojuego de escritorio para un jugador, inspirado en los *juegos de rol* y de *estrategia por turnos*, donde diferentes facciones de enemigos y el propio jugador luchan en un tablero de malla hexagonal por la supervivencia de su grupo, valiéndose de sus habilidades y atributos para entablar batalla o protegerse.

2.1. Introducción: HEX BattleMap

El videojuego *HEX*, en su totalidad, está concebido como un *RPG*. Sin embargo, para este proyecto, la atención se ha focalizado en dar vida únicamente al **sistema de combate**, pues es el entorno donde la inclusión de agentes inteligentes era más interesante (aunque por supuesto, no se descartaría la idea de utilizar las mismas técnicas para agentes que participaran en otros aspectos del juego).

2.1.1. Género y Referencias

El sistema de batalla está claramente influenciado por títulos como *Final Fantasy X* y varias entregas de la saga *Fire Emblem*, pero también mezcla ideas de diseño clave de juegos de mesa como el *Star Wars: Imperial Assault*, en el cual se inspiró su dinámica de acciones y turnos.
Considerando que el juego resultante es una amalgama conceptual de todo esto, es procedente situar a *HEX* en el espectro de *Tactical, Turn-Based RPG*, o *Juego de Rol Táctico Por Turnos*.

2.1.2. Características principales y Propósito

Desde el punto de vista del **jugador**, las batallas en HEX tienen tienen un propósito principal: derrotar a los enemigos en el entorno y ser parte de la última facción en pie.

- En general, la condición de "derrota" se puede reducir a **perder todos los Puntos de Vitalidad (HP)**.
- Así, el grupo de personajes controlados por el jugador debe tratar de derrotar a todos los enemigos a la par que evita ser derrotado.

2.1.3. Estilo Visual

Si uno observa el proyecto y las demos proporcionadas (a día de la entrega de este informe), se dará cuenta rápidamente de que aún no se ha definido un estilo visual para *HEX*. Dado que el enfoque del proyecto permite obviar el apartado estético, las texturas y figuras utilizadas no se corresponden con las de un posible producto comercial en fase final.
Dicho esto, se ha tenido cuidado en mantener una gama de colores lo suficientemente descriptiva como para que la visualización de las demos sea agradable y sencilla cara a las demostraciones.

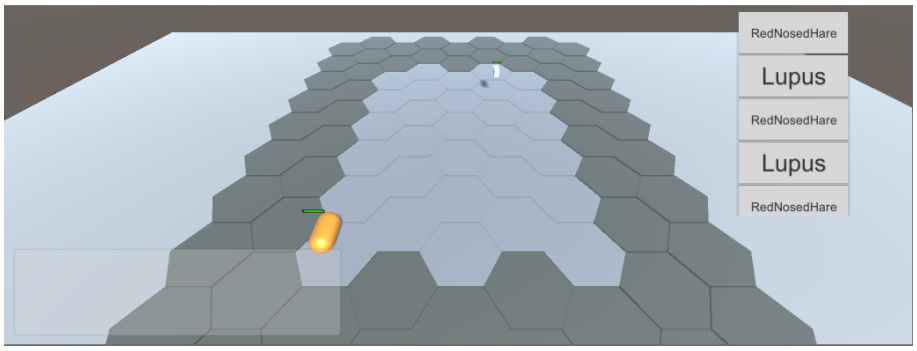

Figura 1: Vista sencilla de la pantalla de juego principal

2.2. Mecánicas de Juego

En esta sección se pretende ilustrar las mecánicas principales del **bucle de juego**. Esto implica explicar el abanico de posibilidades que ofrece el sistema de turnos, las consideraciones a tomar sobre los *personajes jugables* y sobre el *entorno de juego*.

2.2.1. Gameplay

Cuando hablamos de *"gameplay"* nos referimos a la **experiencia de juego:** es decir, cómo el jugador vive el videojuego; la información que recibe y debe procesar, el tiempo que tiene para hacerlo y la forma que tiene de relacionarse con el medio que se le presenta.

El sistema de batalla de **HEX** lo clasificaría como un juego de estrategia por turnos, en el que el jugador vela por la supervivencia de su **Facción** –que, en términos generales, está conformada por las unidades del juego que puede controlar– y combate contra el resto de Facciones hasta que sólo queda una en pie. A grandes rasgos, se

pretende que el jugador **planifique** y disfrute de batallas tácticas en un entorno cerrado contra *agentes inteligentes*, que también tendrán sus estrategias y objetivos propios.

2.2.2. Bucle de juego y Objetivos

El juego comienza con las diferentes Facciones posicionadas sobre el entorno del juego. Después de esto, inmediatamente se les asignan **Turnos** según su parámetro *"Agilidad"* y algunas otras estadísticas relevantes (este proceso se cubre en mayor detalle en la sección 2.2.8) y se toman Turnos hasta que termine la batalla.
Las unidades toman estos turnos y realizan un número determinado de **Acciones**, que depende directamente del tipo de unidad que sean (cada unidad tiene un parámetro *"ACC"* distinto que lo indica).
Las acciones varían entre **moverse, defenderse, atacar o usar habilidades propias** y son el motor principal del transcurso de la batalla. Cuando una unidad lleva a cabo todas las acciones de su turno, se pasa al siguiente y se **vuelven a calcular** (total o parcialmente) **los turnos siguientes**. Los diferentes cursos de acción a elección de la unidad que esté jugando su turno son otro factor importante en el proceso de asignación de turnos posteriores.

El bucle de juego terminará cuando:

- La Facción del jugador elimine a todos los enemigos o sólo quede una Facción *viva*. Una Facción está viva si al menos uno de sus integrantes permanece con vida.
- La Facción del jugador queda incapacitada o es derrotada. En este caso, contará como una victoria para el resto de Facciones enemigas.

Es importante tener en cuenta que, para el entrenamiento de los primeros agentes, no se ha añadido al jugador al bucle de juego. Esto es debido a que, inicialmente, los comportamientos e instintos que se buscan en los NPC no precisan de su colaboración.
A efectos prácticos, esto *elimina la segunda condición de fin del bucle* para las sesiones de entrenamiento propuestas.

A continuación, se puede observar una versión simplificada del **flujo de control** del bucle de juego a modo de esquema:

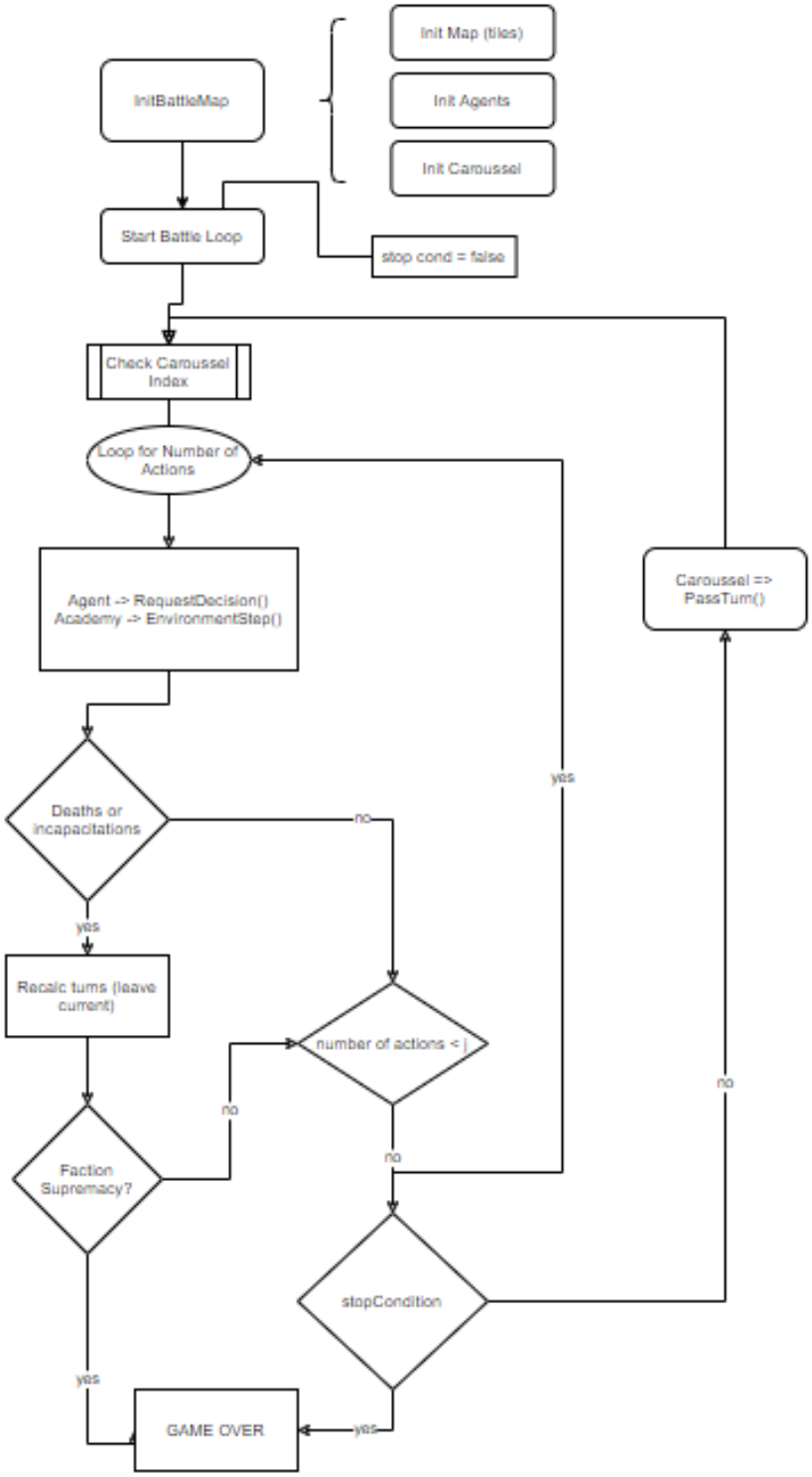

Figura 2: Bucle de juego (código disponible en la sección de algoritmos)

2.2.3. Entorno y Mapas

El entorno en el que se han llevado a cabo simulaciones y entrenamientos es, como ya se había mencionado, una **malla de celdas hexagonales.** El terreno transitable está marcado por el color claro, mientras que las celdas oscuras marcan los **límites del terreno** y los obstáculos.
La forma de los mapas se carga antes de empezar una batalla **desde fichero**, del que también se obtienen las casillas de partida –o *spawns*– para las diferentes Facciones. En las imágenes siguientes se pueden observar (ya cargados en el juego) los diferentes mapas utilizados:

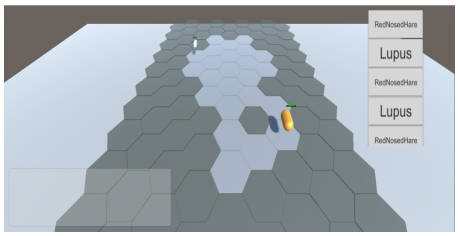

Figura 3: Mapa #1

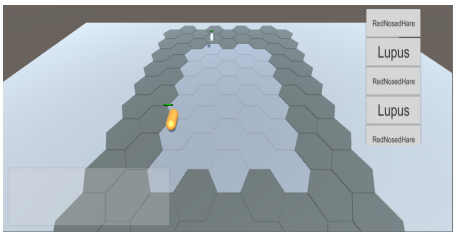

Figura 4: Mapa #2

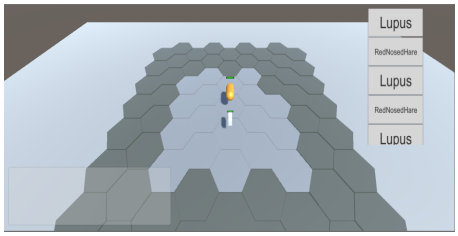

Figura 5: Mapa #3

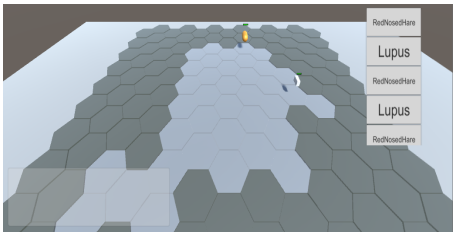

Figura 6: Mapa #4

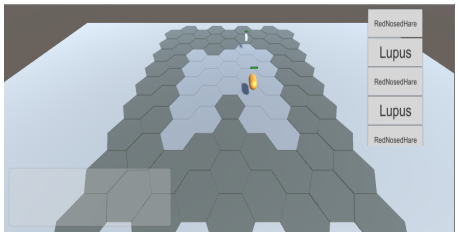

Figura 7: Mapa #5

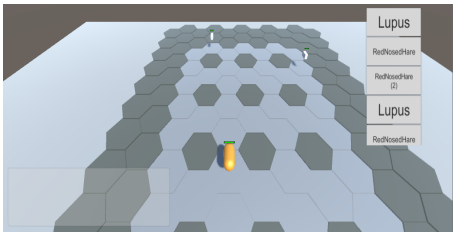

Figura 8: Mapa #6

Los mapas están diseñados para suponer diferentes "retos" para los agentes cara al movimiento, como **evitar moverse hacia obstáculos** y no estancarse en callejones sin salida.

2.2.4. Parámetros de los Personajes

Como es costumbre en los juegos de rol, los personajes y unidades se rigen por una serie de estadísticas o **parámetros**, que controlan el desenlace de todos los eventos que tienen lugar durante la batalla.

Parámetro	**ID**	**Descripción**	Rango numérico de valores
Vitalidad	'HP'	Puntos de vida actuales de la unidad.	[1 - .inf]
Fuerza	'STR'	Factor usado en el cálculo del daño físico realizado por la unidad.	[0 - 255]
Magia	'MAG'	Factor usado en el cálculo del daño mágico realizado por la unidad.	[0 - 255]
Resistencia	'RES'	Factor usado en el cálculo del daño físico mitigado por la unidad al recibir un ataque.	[0 - 255]
Resistencia Mágica	'M.RES'	Factor usado en el cálculo del daño mágico mitigado por la unidad al recibir un ataque.	[0 - 255]
Acciones	'ACT'	Define el número de Acciones que puede realizar una unidad por Turno.	[0 - 6]
Movilidad	'MOV'	Define el número de casillas que puede desplazarse una unidad durante un Turno. Este número es independiente del número de Acciones, sólo restringe su movilidad.	[0 - .undef]
Agilidad	'AGL'	Parámetro que afecta directamente a la frecuencia con la que se le asignan Turnos a una unidad.	[0 - 255]
Precisión	'ACC'	Parámetro que afecta directamente a la probabilidad de que ciertos ataques de la unidad fallen.	[0 - 255]

Cuadro 1: Resumen de Parámetros

Para este videojuego, he decidido no empezar de cero y tomar como referencia los apuntes y cálculos de la comunidad de

Final Fantasy X[5]. Los autores han hecho un trabajo increíble de ingeniería inversa con el que proporcionan una muy buena aproximación al funcionamiento del **cálculo de daño** o el **cálculo de turnos**, entre otras mecánicas importantes del juego.

A efectos prácticos, varias de estas fórmulas son casi directamente extrapolables a mi diseño de juego, así que las he adaptado y utilizado para este proyecto. Para un mayor lujo de detalles en este proceso, se recomienda dirigirse directamente a la *página de Github del proyecto [4].*

2.2.5. Estados Alterados

Otra mecánica importante del juego es la de los **estados alterados**. También como muchos otros juegos de rol, las unidades en el campo de batalla pueden ser afectadas por estos estados –de forma positiva o negativa– ocasionando potencialmente grandes cambios en el transcurso de la batalla.

Cabe destacar que, a diferencia de los parámetros, normalmente los estados alterados son **temporales:** es decir, solo aplican su efecto durante un número limitado de turnos.
Además, no son **predefinidos:** o lo que es lo mismo, se aplican puntualmente a causa de una serie de acciones específicas o por el efecto de alguna habilidad –esto, a excepción de ciertas unidades que puedan contar con el efecto al inicio de la batalla o de *manera permanente* por alguna circunstancia especial.

Cada tipo de estado alterado tiene su variante **positiva**, que mejora las características o el rendimiento de la unidad afectada de alguna forma; y su contraria, la **negativa**, que por supuesto supone una penalización.
En cuanto a la **duración** de los estados, por otro lado, tenemos un tiempo base de **10 turnos**, que puede aumentar o disminuir circunstancialmente por obra de alguna habilidad o evento.

Algunas de las demás restricciones asociadas a estos cambios de estado serán nombradas en la siguiente sección para mayor simplicidad y organización del contenido.
A continuación, como en el apartado anterior, tenemos una lista de los estados alterados con sus efectos en combate:

Estados alterados	ID	Descripción
Bravura/AntiBravura	BRAVERY	Aumenta/Disminuye el daño físico inflingido a otras unidades en un factor de x1.5/x0.5
Fe/AntiFe	FAITH	Aumenta/Disminuye el daño mágico inflingido a otras unidades en un factor de x1.5/x0.5
Armadura/AntiArmadura	ARMOR	Disminuye/Aumenta el daño físico recibido en un 50 %
Escudo/AntiEscudo	SHIELD	Disminuye/Aumenta el daño mágico recibido en un 50 %
Prisa/Freno	HASTE	Aumenta/Disminuye la frecuencia en la que se le asignan turnos a la unidad en un 50 %
Regeneración/Veneno	REGEN	Recupera/Pierde un porcentaje de la vida máxima (HP) por turno hasta que expire el efecto

Cuadro 2: Resumen de Estados Alterados

2.2.6. Reglas y Restricciones

Una vez introducidos parámetros y estados, es conveniente abordar las restricciones y normas asociadas al transcurso de la batalla, a modo de una estructura/resumen de las **reglas** del juego que se pueda consultar *a posteriori* con comodidad:

- El bucle de juego **finaliza** cuando:
  - La Facción del jugador elimina al resto de enemigos.
  - La Facción del jugador queda eliminada.
  - En caso de dejar al jugador fuera del escenario inicial (escenarios cubiertos en el proyecto), cuando sólo una Facción queda en pie.
- Una Facción **queda eliminada** cuando todas las unidades que la componen **mueren**.
- Una unidad **muere** cuando pierde todos sus Puntos de Vida (HP).
- Los **Turnos** se reparten constantemente mientras dure la batalla. Únicamente las unidades **vivas** reciben turnos.
- Las unidades mantienen una **posición** en el mapa, pudiendo cambiarla a través del movimiento. Existe una restricción

numérica del número de casillas que una unidad se puede mover en un Turno y está representada por su parámetro *MOV*.

- Dos unidades no pueden estar en la misma casilla al mismo tiempo.
- El terreno franqueable del mapa **está claramente señalado** en color más claro. El resto de casillas son **obstáculos** intransitables y son útiles para marcar los límites del tablero.
- Durante un **Turno**, una unidad puede realizar un número de **Acciones** igual a su parámetro *ACT*. Una acción puede ser dedicada a cosas como **atacar, defenderse, usar una habilidad** o **moverse**, si el parámetro *MOV* lo permite.
- El límite de movimientos posibles para una unidad se puede **cumplir de manera intermitente**: por ejemplo, moverse una casilla, atacar, moverse otra casilla (asumiendo **MOV mayor o igual a 2** y **ACT igual a 3**).
- Los **parámetros**, a excepción de los Puntos de Vida (HP), son invariables para todas las unidades durante el transcurso de la batalla.
- Los **estados alterados** (en su variante positiva y negativa), sin embargo, pueden provocarse durante la batalla. Su duración base es 10 turnos, calculados sobre la unidad afectada.
- Los estados alterados perjudiciales y los beneficiosos pueden provocarse sobre su contrario para **cancelarlo**. Esto regresará a la unidad afectada a un estado neutro.
- Un estado alterado **puede volver a provocarse sobre sí mismo**, reiniciando así el tiempo de duración base. Sin embargo, no puede provocarse el mismo estado alterado dos veces al mismo tiempo sobre la misma unidad.
- Los estados periódicos como **el veneno** o **regeneración** hacen su efecto justo *al final* del turno del afectado.

2.2.7. Habilidades

Las **habilidades** son una parte importante del juego pues, en el sentido práctico, componen la totalidad del abanico de acciones disponibles para los agentes. Algunas habilidades base son **atacar, defender** o **moverse** y son comunes para prácticamente todas las unidades. Sin embargo, existen otras habilidades, con efectos diversos, que dependen directamente del tipo de unidad: es uno de los factores que le da variedad al juego e incita a que diversos personajes sigan diferentes estrategias en combate.

Cada habilidad tiene además asignado un **Rango**, que básicamente viene a indicar un *coste de tiempo* asociado a dicha acción. A mayor Rango, mayor es también el coste temporal de la habilidad y más se tardará en volver a asignar un turno a la unidad que la use.
Los Rangos oscilan entre 0 y 10, siendo el rango estándar el *3*. Atacar, Moverse y Defenderse tienen rango 3 y un efecto promedio en el desarrollo de la batalla; sin embargo, habilidades más poderosas tendrán mayores costes, por lo general. Esto pretende crear un balance **riesgo/recompensa** que no premia excesivamente usar habilidades poderosas constantemente.

Otra cuestión importante es el hecho de que una unidad pueda usar más de una habilidad por Turno (de hecho, ejecutará una por cada acción que tenga disponible). Entonces, ¿cómo determinaríamos el Rango asignado a dicho Turno?
En las fases tempranas de diseño se plantearon varias opciones, pero finalmente se optó por utilizar el Rango de la habilidad más costosa, pues parecía la métrica más "justa" para unidades con **poca varianza** en los Rangos de sus habilidades disponibles.

2.2.8. El "Caroussel" de Turnos

Otra parte en la que vale la pena reparar es en el proceso de **Cálculo de Turnos**, un pilar importante en la infraestructura del videojuego (código disponible en la sección de algoritmos al final de la memoria).

El **método de cálculo de turnos** está basado en el sistema **CTB** o *Clock Timed Battle* ya presente en algunos JRPG como FFX o FF Tactics, en los que todo gira alrededor de los "ticks" de cierto **reloj interno**: todas las unidades en el entorno cuentan con un **Contador**, que disminuye en 1 unidad cada vez que este reloj da

un "tick", y reciben su turno cuando este llega a 0.
Cuando se toma un turno y al **inicio de la partida**, el Contador se reinicia a un valor que depende de tres factores principales: el estado **Prisa/Freno,** el **Rango** de la última habilidad utilizada y la **Agilidad** –aunque este último no se utiliza por si solo, sino a través de un índice llamado *TickSpeed* que se calcula desde su valor. El valor inicial del Contador o **ICV** sigue la fórmula siguiente:

$$ICV(InitialCounterValue) = Haste \times TickSpeed \times LastSkillRank$$

El **"Caroussel"** de Turnos contiene siempre los **16 turnos siguientes** incluyendo al actual. Para calcular en adelantado, se asume que las unidades asignadas usarán habilidades de rango 3.
Esto implica **mantener los Contadores y el número de turnos restantes en estado Prisa/Freno** por separado para todas las unidades, ya que de otro modo sería problemático realizar el cálculo constantemente. Todo esto tiene sentido porque, dados determinados eventos en el bucle de juego, se deberán **re-calcular** todos los turnos una vez más –principalmente, **cuando una unidad muera** o cuando **se use una habilidad de Rango distinto a 3**. En caso contrario, simplemente se calculará el turno siguiente usando los cálculos previos hechos por adelantado.

Así pues, este sería un buen resumen de los pasos que toma el algoritmo de asignación de turnos:

- Al principio de la partida, se calculan los 16 primeros turnos de la batalla y se mantienen los contadores procedentes para cada unidad viva.
- Cuando se pasa un turno, en función de los eventos que hayan tenido lugar durante el mismo, se toma una decisión:
  - Se calcula el siguiente turno de manera normal.
  - Se calcula de nuevo todo el Caroussel de Turnos y se almacenan los nuevos 'valores futuros' de los contadores.
- Para asignar un turno, se dan "ticks" del Reloj Interno hasta que algún Contador llegue a 0 y se asigna el turno. Una vez asignado, se reinicia su Contador y se repite hasta que se llegue al cupo de turnos pre-calculados.
- En caso de que **dos Contadores lleguen a 0 al mismo tiempo**, se sigue una política *First-Come,First-Served* para distribuirles sus turnos.

3. NPC e Inteligencia Artificial

Llegamos a la parte fundamental de este proyecto, en la que se pasará a describir el diseño de los NPC: los **agentes inteligentes** que han sido entrenados. Se cubrirán para cada uno las consideraciones sobre su comportamiento, propósitos iniciales, el **enfoque heurístico** que aproxima su comportamiento deseado y las **observaciones** que reciben durante el entrenamiento.

3.1. Jerarquía de clases

Se puede utilizar conceptos propios de la Programación Orientada a Objetos para generalizar ideas como *"Unidad de Batalla"*, *"Enemigo/NPC"* o *"Personaje Jugable"*, estableciendo así una **jerarquía**, modo del que conviene entender la **estructura** organizativa de clases que comparten las "Unidades de Batalla" o *"GameCharacters"*. Dicha estructura se puede ver a continuación, (las flechas indican relaciones de herencia simple):

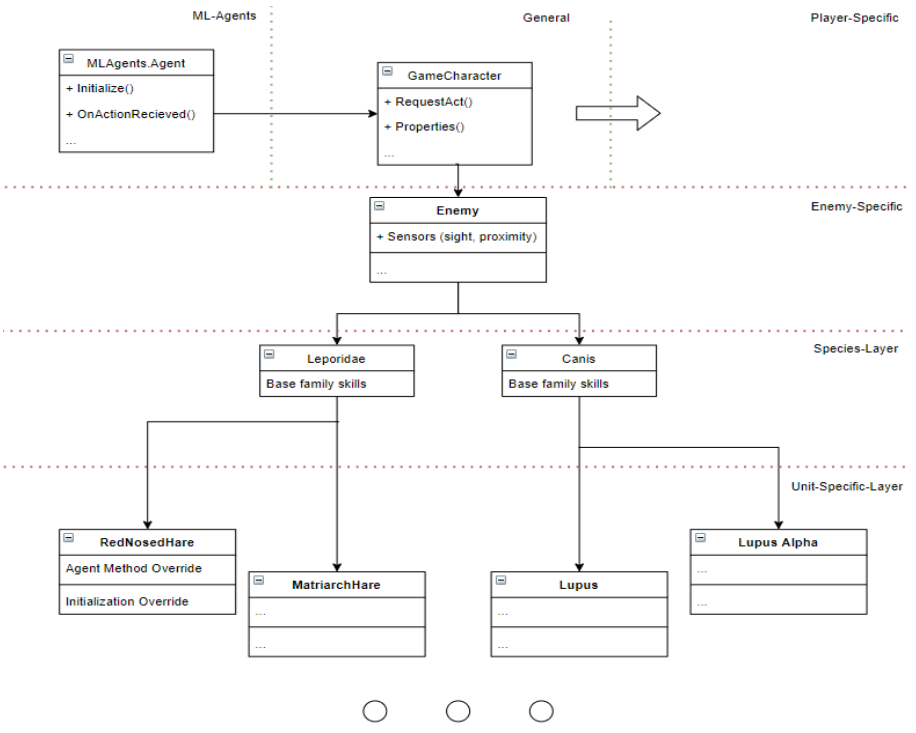

Figura 9: Diagrama de Clases Sobre-simplificado de Agentes

El árbol está a su vez dividido en secciones lógicas que indican el

nivel de abstracción. Está inspirado directamente en un **árbol genealógico de especies** y se puede ver de la misma manera.

3.1.1. *GameCharacter*

GameCharacter es la clase base (quitando MLAgents.Agent, su clase padre) de la que heredan todos los personajes del juego. Es algo así como el marco que un agente tiene que seguir para poder participar en el bucle de juego como los demás. Desde esta clase se llaman a varios métodos que funcionan del mismo modo para todas las unidades, como pueden ser el **tomar una acción** en batalla cuando el juego lo precise o actualizar atributos comunes.

También contienen muchas de las **propiedades** y atributos comunes a todas las unidades de batalla, como pueden ser los **parámetros y estados alterados** y sus métodos de acceso.

3.1.2. Enemigos

La clase *Enemy* es específica para los NPC y contiene todos los métodos útiles para percibir eventos y unidades a su alrededor. Esta "percepción" no se limita al sentido físico de la palabra: los NPC deben ser capaces de **distinguir la facción** a la que pertenecen las unidades que le rodean, conocer la **distancia que les separa** y saber si tienen **presas o aliados adyacentes**.

Todos los **sensores** utilizados para las percepciones en los agentes parten de esta clase, pues son métodos generales que se pueden reutilizar en todo tipo de situaciones para diferentes patrones de estrategia –aunque se da la opción de sobrescribir algunos de estos métodos desde las clases hijas.

3.1.3. Personajes Jugables

A día de la realización de este informe, los personajes jugables no están implementados y tampoco forman parte de los entrenamientos con agentes. Sin embargo, la única distinción real que hacen los agentes es entre *"presa"* y *"depredador"*, lo que incluye indirectamente al jugador en sus consideraciones.
Esto es así para propiciar que los agentes entrenados en un entorno sin jugadores **actúen según lo aprendido** una vez incorporados al juego y **sin necesidad de nuevos entrenamientos** específicos; su conducta hacia el jugador dependerá de si lo ven como un objetivo o como una amenaza.

Todos los agentes diseñados hasta el momento tratan a la facción del jugador en calidad de "presa", pues en general, de esta manera entenderán que deben atacarle dada la oportunidad.

3.2. Especies / Familias

Tal y como se mencionaba antes, el árbol de clases podría fácilmente entenderse como un árbol de especies común. Las Especies o **Familias** en HEX representan agrupaciones de enemigos del *mismo tipo*: comparten características, metas, **depredadores y presas.** Por norma general, todos los miembros de una Familia son depredadores natos de las mismas Especies y deben protegerse de las mismas Familias enemigas –es una característica que los define. De este modo, todos los miembros de la Especie *Canis* serán depredadores para las unidades de la Familia *Leporidae*, por ejemplo.

Desde el punto de vista de diseño del juego, es posible que las Familias compartan otros atributos, como habilidades específicas o propiedades. Por otro lado, en lo que respecta a sus conductas, se puede decir que los propósitos de las unidades de una misma Especie son similares y, por tanto, la implementación y las recompensas que reciben por determinadas acciones compartidas serán similares también –y a veces, completamente idénticas.

3.2.1. Leporidae

En la realidad, **Leporidae** es el nombre científico para la especie de los conejos y las liebres. En HEX, ésta será la clase para la Familia con el rol de "presa" en los entrenamientos, concordando también así con el imaginario colectivo.
Los agentes de la Familia Leporidae están diseñados con la idea de 'sobrevivir todo el tiempo posible' en mente. Se pretende que sean enemigos relativamente pasivos, contentos con simplemente corretear por el mapa. También se plantearon en las fases iniciales que funcionaran más como una *"colonia matriarcal"*, donde los agentes más sencillos prioricen la vida de la figura 'madre' de la colonia, aunque este es un comportamiento que por el momento no se ha conseguido replicar.

Sus **depredadores naturales** son los de la Familia *Canis*, y sus **presas** son la Facción del jugador y la Familia *Abomination*.

3.2.2. Canis

Canis es un género de mamíferos placentarios de la familia *Canidae* en la realidad. Para este proyecto, los Canis toman el papel de "depredadores" en el bucle de juego.
Su diseño es bastante sencillo: *'cazar a las presas para ganar la partida'*. Se pretende que persigan a sus objetivos por el mapa, atacando de manera activa para vaciar sus barras de vida. En fases iniciales se planteó también que funcionaran como una *"manada"* donde hubiera un *líder* que inspirara al grupo mientras el resto colabora para hacer el trabajo sucio, aunque esto es algo que no se ha conseguido replicar en los entrenamientos.

Sus **depredadores naturales** son de la Familia *Abomination* y sus presas son el jugador y la Familia *Leporidae*. Estas relaciones completan un 'triángulo de clases' y pretenden aportar cierto balance al ecosistema resultante.

3.2.3. Abomination

Esta Familia es parte del contenido teórico del diseño de juego que no ha llegado a recibir una implementación por el momento.

Se supone que los miembros de esta Especie serían entes caóticos de gel y masa poco reconocible. Algo así como los típicos *slimes* de los juegos de rol. Los Lupus les temen, mientras que las Leporidae han evolucionado para poder consumir sus cuerpos.

Así, sus principales depredadores serían de la Familia *Leporidae* y sus presas el jugador y la Familia *Lupus*.

3.3. Sensores

Antes de pasar a describir los agentes implementados por separado, vale la pena explicar cómo funcionan los **Sensores** que les ayudan a observar el entorno.

Los agentes utilizan diferentes métricas para entender el entorno: **valores numéricos de parámetros** (los HP restantes, por ejemplo) y **distancias** hasta depredadores o unidades cercanas (medidas usando como referencia algunas de las fórmulas en [5]). Sin embargo, los agentes también necesitan algún modo de entender el medio en el que viven y sus características, los factores que puedan impedir su movimiento, obstáculos y la posición relativa de otras unidades en el entorno.
Sumado a esto, ha habido que representar estas observaciones de un modo que las redes neuronales puedan entender como *input*. Aquí es donde entran estos sensores que, a fin de cuentas, no son sino un medio para **condensar esta información espacial**.

3.3.1. Sensor de Proximidad

Lo que llamaremos **Sensor de Proximidad** es una herramienta que indica al agente las unidades que se encuentran dentro de su rango de visión a **cada una de las direcciones generales de movimiento**.
Las direcciones están codificadas por **índices** tanto cara al movimiento como a los sensores (tal y como se puede ver en la fig 10) por lo que el sensor no es más que una caja negra que devuelve un **vector de 6 números en formato punto flotante**, cada uno correspondiéndose con una dirección. Los números son calculados en base a los *depredadores* y *presas* que se encuentren en cada dirección y representan un **factor de la recompensa que otorga moverse hacia una dirección concreta**. El código correspondiente se encuentra en la sección de algoritmos, al final de la memoria.

3.3.2. "Sensor de Adyacencia"

Para los peligros y recompensas más inmediatos, sin embargo, tenemos el **Sensor de Adyacencia**. Al contrario que el de proximidad, este sensor también controla las condiciones del terreno y **avisa al agente, entre otras cosas, de si una casilla adyacente es transitable**. Este sensor proporciona un **valor único para**

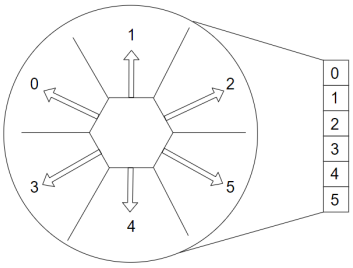

Figura 10: Direcciones Generales y Vector Sensor

cada dirección que depende directamente del estado de la casilla: *0.5* cuando hay una presa en esa dirección, *-1.0* cuando hay un obstáculo y otros valores para observaciones diferentes.
El código también está a su disposición en la sección de algoritmos, al final de la memoria, junto al del sensor de proximidad.

3.3.3. "Sensores de Distancia"

En esencia, este es el tipo de sensor más sencillo. Retorna un valor numérico que representa la distancia entre una unidad concreta y el agente que recibe la información. En los agentes propuestos, se utilizan principalmente dos sensores de este tipo:

- El Sensor de **Distancia hasta el depredador más cercano**.
- El Sensor de **Distancia hasta el aliado más cercano**.

Dado que los agentes propuestos son relativamente sencillos, la toma de decisiones prioriza los peligros y recompensas más inmediatos, aunque se podría, por supuesto, crear sensores más sofisticados a partir de estos dos.

3.4. Estado de los NPC

En las siguientes secciones se mostrarán las fichas técnicas de los NPC diseñados para HEX y para este proyecto. Esto incluye **parámetros, acciones disponibles, habilidades, comportamientos heurísticos, funciones de recompensa** y **observaciones** de los agentes.

Todos estos datos corresponden con las **versiones finales de los agentes**, a día de la finalización de esta memoria del proyecto.

3.5. Ficha técnica de Agente: *Lupus*

Lupus (modelo *in-game* en la figura 11) es el NPC más básico de la Familia Canis y el primer agente creado en las fases tempranas de desarrollo.

Parámetros:

HP	ATK	MAG	RES	M.RES	ACT	MOV	AGL	ACC
78	7	1	21	1	2	2	59	0

Vector Acción:

- **Tamaño:** 2
- **[0]:** Habilidad seleccionada.
- **[1]:** Dirección (parámetro que toma la habilidad a usar).

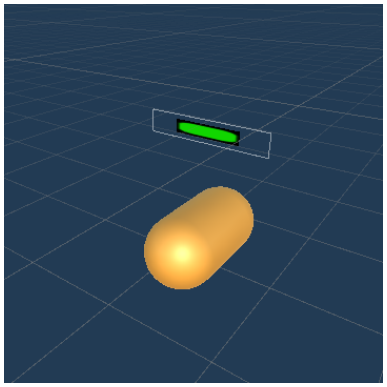

Figura 11: Modelado de Lupus

3.5.1. Comportamiento y Heurísticas

Su conducta es la representación más pura del concepto de 'cazar para vivir', pues es su única meta real.

Su comportamiento programado consiste en moverse aleatoriamente mientras no se tengan presas a la vista; **si se detecta una presa potencial**, la perseguirá hasta poder atacarla, acción que repetirá hasta que uno de los dos pierda todos sus HP.
El agente priorizará a las presas más cercanas y atacará a las demás cuando acabe con la presa actual.

3.5.2. Función de recompensa

Lupus es un agente diseñado para ser **agresivo**, por lo que se le dan incentivos por atacar y aniquilar a sus presas. No le debería importar recibir daño si puede ocasionarlo (aunque tampoco le gusta perder).

- **Feedback Pasivo**:
  - Negativo: Recibe una penalización de -0.1 cada vez que toma una acción. Esto se hace para incentivar la pro-actividad y para que intente ganar lo más rápido posible.
- **Al atacar**:
  - Positivo: Cuando ataca a una presa +1.0
  - Negativo: Cuando ataca al aire -0.5; Cuando *no* ataca a una presa -1.0
- **Al moverse**:
  - Negativo: Cuando intenta moverse hacia un obstáculo o hacia una casilla ocupada -0.5
- **Al defenderse**:
  - Negativo: Siempre que defiende -0.01
- **Cuando recibe daños**: -0.3
- **Cuando su facción es derrotada**: -5.0
- **Cuando su facción gana**: +5.0

3.5.3. Observaciones

El agente recibe la información siguiente:

- **Valor numérico** de su parámetro HP.
- El **Sensor de Adyacencia**
- El **Sensor de Proximidad**
- La **distancia (int)** que le separa de su presa más cercana.
- El **número de aliados** que quedan vivos junto a él.

El **número de aliados** es una observación que, al principio, no se utilizaba. Sin embargo, resulta útil para entornos de entrenamiento más complicados, donde puede haber más de un Lupus en el campo de batalla.

3.6. Ficha técnica de Agente: *Lupus Alpha*

Este agente está implementado, pero no se han llegado a hacer entrenamientos para él. Esta sección es un enfoque teórico del mismo.

Lupus Alpha es el cabecilla de la manada de Lupus y, como tal, cuenta con parámetros y habilidades superiores al resto. Concretamente, es capaz de usar la habilidad **"Aullido"**, que inspira y fortalece a sus aliados, dándoles el estado **Bravura** si están en alguna de sus casillas adyacentes. Contar con una habilidad de bonificación lo convierte en un factor decisivo en la batalla y multiplica el poder de los Canis que estén en sus alrededores.

Parámetros:

HP	ATK	MAG	RES	M.RES	ACT	MOV	AGL	ACC
128	11	2	21	10	2	2	65	0

Vector Acción:

- **Tamaño:** 2
- **[0]:** Habilidad seleccionada.
- **[1]:** Dirección (parámetro que toma la habilidad a usar).

3.6.1. Comportamiento y Heurísticas

Ser el jefe de la manada viene con diferentes responsabilidades a las de ser un simple cazador. Sí bien conserva su sed de sangre y atacará y perseguirá a las presas cercanas junto con los demás Lupus, el agente está diseñado para hacerlo desde un enfoque más 'reservado'. Lupus Alpha **se centrará en usar su 'Aullido'** cuando haya aliados cerca cada cierto tiempo y **priorizará esta acción y el estar cerca de sus aliados** sobre perseguir a las presas per sé. Además, intentará protegerse cuando su vida baje por debajo del 30 %.

3.6.2. Función de recompensa

Lupus Alpha está diseñado para funcionar como soporte de los demás Canis en batalla, pero conserva los rasgos agresivos que caracterizan a la raza.

- **Feedback Pasivo**:
  - Negativo: Recibe una penalización de -0.1 cada vez que toma una acción. Esto se hace para incentivar la pro-actividad y para que intente ganar lo más rápido posible.
- **Al atacar**:
  - Positivo: Cuando ataca a una presa +1.0
  - Negativo: Cuando ataca al aire -0.5; Cuando *no* ataca a una presa -1.0
- **Al moverse**:
  - Positivo: Relacionado con la distancia hasta el aliado más cercano
  - Negativo: Cuando intenta moverse hacia un obstáculo o hacia una casilla ocupada -0.5
- **Al usar 'Aullido'**:
  - Negativo: Cuando lo lanza sin tener aliados cerca o **antes de que pase cierto tiempo** desde la última vez que usó la habilidad.
  - Positivo: +0.5 x número de aliados que acierte.
- **Al defenderse**: -0.1 (+0.1 si tiene menos del 30 % de su vida máxima).
- **Cuando recibe daños**: -0.3
- **Cuando su facción es derrotada**: -5.0
- **Cuando su facción gana**: +5.0

3.6.3. Observaciones

El agente recibe la información siguiente:

- **Valor numérico** de su parámetro HP.
- El **Sensor de Adyacencia**
- El **Sensor de Proximidad**
- La **distancia (int)** que le separa de su aliado más cercano.
- El **número de aliados** que quedan vivos junto a él.
- El **tiempo transcurrido** (en función del número de turnos) desde la última vez que usó la habilidad 'Aullido'.

3.7. Ficha técnica de Agente: *Red Nosed Hare*

Red Nosed Hare (modelo *in-game* en la figura 12) es el NPC más básico de la Familia Leporidae y funcionará como la *presa tipo* para Lupus.

Parámetros:

HP	ATK	MAG	RES	M.RES	ACT	MOV	AGL	ACC
78	3	2	21	10	1	1	65	0

Vector Acción:

- **Tamaño:** 2
- **[0]:** Habilidad seleccionada.
- **[1]:** Dirección (parámetro que toma la habilidad a usar).

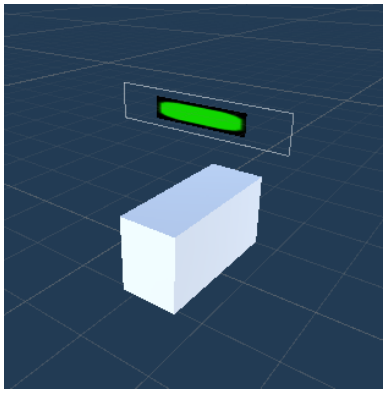

Figura 12: Modelado de Red Nosed Hare

3.7.1. Comportamiento y Heurísticas

Si Lupus es el agente diseñado para ser agresivo y acechar a sus presas, Red Nosed Hare es todo lo contrario. Su única misión en la vida es **sobrevivir** el mayor tiempo posible, y eso implica huir de sus depredadores. Red Nosed Hare huirá de los depredadores que detecte en batalla, intentando alargar su vida el mayor tiempo posible hasta que lo capturen o se quede sin lugar al que huir.
La **función heurística** que controla su comportamiento le hace huir de su depredador más cercano (elige la dirección contraria a la que este se encuentra, de ser posible).

3.7.2. Función de recompensa

Red Nosed Hare es un agente pasivo y sus recompensas van orientadas principalmente a la supervivencia y a la huída:

- **Feedback Pasivo**:
  - Positivo: Recibe una recompensa de +0.1 por cada acción que toma. Esto consigue que el agente quiera permanecer vivo el mayor tiempo posible.
- **Al atacar**:
  - Positivo: Cuando ataca a un depredador +0.2
  - Negativo: Cuando ataca al aire -0.5; Cuando *no* ataca a un depredador -1.0
- **Al moverse**:
  - Positivo: Proporcional a la distancia hasta su depredador más cercano.
  - Negativo: Cuando intenta moverse hacia un obstáculo o hacia una casilla ocupada -0.5
- **Al Defenderse**:
  - Negativo: Siempre recibe un -0.1 (prefiere huir).
- **Cuando recibe daños**: -0.5
- **Cuando su facción es derrotada**: -5.0
- **Cuando su facción gana**: +5.0

3.7.3. Observaciones

El agente recibe la información siguiente:

- **Valor numérico** de su parámetro HP.
- El **Sensor de Adyacencia**
- El **Sensor de Proximidad**
- La **distancia** que le separa de su depredador más cercano.
- El **número de aliados** que quedan vivos junto a él.

3.8. Ficha técnica de Agente: *Matriarch Hare*

Este agente está implementado, pero no se han llegado a hacer entrenamientos para él. Esta sección es un enfoque teórico del mismo.

Matriarch Hare es una versión mejorada de Red Nosed Hare y la figura materna de su comunidad. Si Lupus tiene a Lupus Alpha, las liebres tienen a este agente para equilibrar la balanza de poder. Esta unidad está enfocada en la protección de sus aliados y cuenta con dos habilidades defensivas: *'Abrazo Materno'* otorga el estado Armadura a los aliados adyacentes (funciona como la habilidad de Lupus Alpha) y *'Provisiones'* le permite curar a un aliado.

Parámetros:

HP	ATK	MAG	RES	M.RES	ACT	MOV	AGL	ACC
128	7	2	21	10	2	2	65	0

Vector Acción:

- **Tamaño:** 2
- **[0]:** Habilidad seleccionada.
- **[1]:** Dirección (parámetro que toma la habilidad a usar).

3.8.1. Comportamiento y Heurísticas

Matriarch Hare es el apoyo perfecto que necesita un grupo de Red Nosed Hare para sobrevivir frente a una manada de Canis con Lupus Alpha. Su misión es permanecer cerca de sus aliados para protegerlos con el estado Armadura e ir curando a las unidades heridas a su alrededor. Su mayor capacidad de movimiento, además, le permite reposicionarse ante amenazas, pero seguirá en aprietos si se ve rodeada y superada en número.
Su **comportamiento heurístico** configura una estrategia sencilla: permanecer curando y protegiendo a los demás Leporidae mientras sea posible y solo atacar a sus depredadores cuando no le quede otro remedio.

3.8.2. Función de recompensa

Matriarch Hare funciona también como soporte de los Red Nosed Hare en batalla y tratará siempre de mantener al grupo vivo.

- **Feedback Pasivo**:
  - Positivo: Recibe una recompensa proporcional al número de agentes de la familia Leporidae vivos en su turno.
- **Al atacar**:
  - Positivo: Cuando ataca a un depredador +0.2
  - Negativo: Cuando ataca al aire -0.5; Cuando *no* ataca a un depredador -1.0
- **Al moverse**:
  - Positivo: Inversamente proporcional a la distancia hasta su aliado más cercano.
  - Negativo: Cuando intenta moverse hacia un obstáculo o hacia una casilla ocupada -0.5
- **Al usar la habilidad 'Provisiones':**
  - Positivo: Si cura a un aliado +1.0
  - Negativo: Si cura a un enemigo -1.0 o la lanza al aire -0.5
- **Al usar la habilidad 'Abrazo Materno':**
  - Negativo: Cuando lo lanza sin tener aliados cerca o **antes de que pase cierto tiempo** desde la última vez que usó la habilidad.
  - Positivo: +0.5 x número de aliados que acierte.
- **Al Defenderse**: -0.1 (siempre)
- **Cuando recibe daños**: -0.5
- **Cuando su facción es derrotada**: -5.0
- **Cuando su facción gana**: +5.0

3.8.3. Observaciones

El agente recibe la información siguiente:

- **Valor numérico** de su parámetro HP.
- Los sensores de **Adyacencia** y **Proximidad**.
- La **distancia** que le separa de su depredador y de su aliado más cercano.
- El **número de aliados** que quedan vivos junto a él.
- El **tiempo transcurrido** (en función del número de turnos) desde la última vez que usó la habilidad 'Abrazo Materno'.

4. Proceso de Desarrollo y Entrenamiento

En esta sección se pretende discutir en profundidad los experimentos y entrenamientos realizados con el set de agentes previamente presentado. Se expondrá de manera secuencial, tomando los resultados relevantes y/o interesantes para contextualizar las pruebas realizadas y sus respectivas motivaciones. Se hará un énfasis en la metodología y en los cambios de parámetros de entrenamiento en cada fase.

4.1. El Juego y el *"Modo Entrenamiento"*

El enfoque principal de este proyecto está en entrenar a los agentes inteligentes para que funcionen en un entorno de juego real. Sin embargo, cabe mencionar antes de comenzar con la visión general de los entrenamientos que, el entorno en el que se han realizado los mismos está *completamente aislado* del juego real.

Es una práctica común en este tipo de casos, pero la escena de Unity en la que han vivido estos agentes es una **réplica del escenario real y del bucle de juego** ajustada para el método de entrenamiento de *MLAgents*.

Esto supone que se hayan arreglado las escenas para poder **ejecutar varias instancias del juego en paralelo** desde el editor de Unity. Es posible entrenar desde ejecutables de Unity directamente, por lo que hacerlo desde el editor no es estrictamente necesario. Sin embargo, es por ahora el método preferible, a mi criterio, dado el estado actual del videojuego.
Hacerlo así, además, ha facilitado infinitamente el proceso de depuración de la aplicación en fases tempranas, por lo que se puede considerar incluso una buena práctica en su contexto.

Para futuras versiones, sin embargo, y para facilitar la visualización de los mapas de manera individual, he añadido la opción de poder **alternar entre las diferentes cámaras** de la escena (cuando hay varias instancias ejecutándose a la vez en la misma), lo que permite observar los patrones de conducta y cómo van evolucionando con más detalle incluso desde un ejecutable.

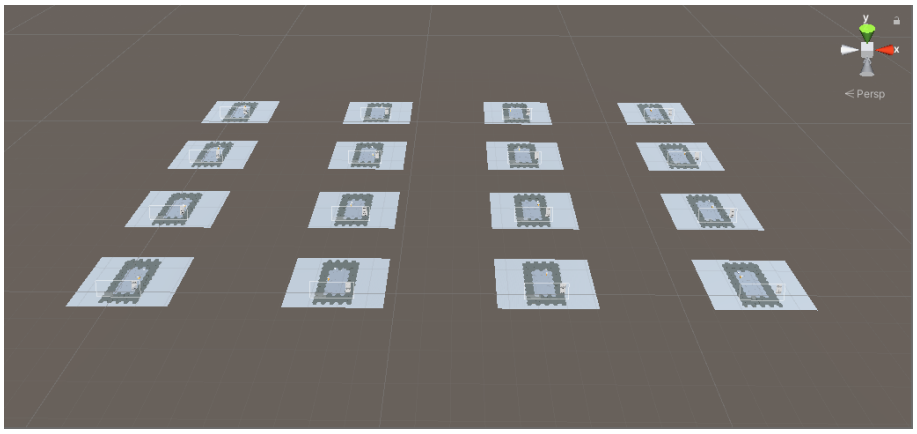

Figura 13: Vista aérea de la escena "Modo Entrenamiento" con 16 instancias funcionando en paralelo (tomada desde el inspector de Unity).

4.1.1. Entrenando con Unity MLAgents

Antes de pasar al proceso de desarrollo y al análisis de los entrenamientos, conviene aclarar un par de conceptos que tienen que ver con el kit de MLAgents y facilitarán la comprensión de sus aspectos más técnicos:

- **Agente:** Unido al *GameObject* de un NPC en Unity, es el componente encargado de gestionar las observaciones y decisiones que tome dicha unidad durante la partida, además de las recompensas recibidas.
- **Comportamiento:** Los comportamientos van unidos a los agentes. Se pueden pensar como funciones únicas y distinguibles por su **nombre**, que reciben observaciones y recompensas del agente y devuelven acciones.
- **Academia:** El objeto *Academy* dentro del juego. Es la encargada de controlar el entorno y de gestionar los entrenamientos, entre otras cosas.
- **Cerebro:** Es la red neuronal encargada de procesar la información recibida por el agente para posteriormente traducirla en acciones. El objetivo real de los entrenamientos es el de entrenar a los cerebros de los diferentes agentes para que desarrollen comportamientos acordes a sus necesidades.
- **PPO:** O *Proximal Policy Optimization*, es el algoritmo subyaciente utilizado para ajustar las redes neuronales durante los entrenamientos.

4.2. Primeros Pasos: Adaptar el Juego a MLAgents

El hecho de que el sistema de batalla por turnos en HEX lleva a usar **toma de decisiones bajo demanda** en los agentes. Esto significa que los agentes no tomarán decisiones cada cierto número de *Updates* en Unity –y por tanto, tampoco tendrán un componente *ActionRequester* que se encargue de solicitar las decisiones al cerebro cada cierto tiempo.
En nuestro caso, lo ideal sería poder **controlar esta toma de decisiones** enteramente desde el bucle de juego, pudiendo pedirle a los agentes que actúen solamente cuando les llegue su turno. Para conseguir esto, debemos asegurarnos de tres cosas en nuestro código:

- Primero, de cambiar el valor del booleano *Academy.Instance.AutomaticSteppingEnabled* a **false**. Esto evita que los agentes intenten tomar decisiones por su cuenta.
- Después, llamar desde el bucle de juego, cuando sea necesario, al método *Agent.RequestDecision*. En nuestro caso, se llama indirectamente a través del *GameCharacter.RequestAct*.
- Inmediatamente después, llamar a la función **AcademyStep** para que el agente cuya decisión ha sido solicitada tome la acción correspondiente y se dé un 'paso' desde el punto de vista de la Academia.

Ahora solamente habría que gestionar la **condición de parada** del entorno. En otras palabras, el evento que indica el fin de un **Episodio de la Academia** (lo equivalente a una partida) y el comienzo del siguiente. Aquí es donde se encontró el **primer gran problema de adaptación**.
Aunque en los documentos oficiales se apunta a que debería llamarse a la función *Academy.EnvironmentReset* automáticamente cuando se señalase el final del episodio de cada uno de los agentes, lo cierto es que esto no sucedía. Otras alternativas propuestas en foros estaban **obsoletas** o no tenían sentido en la versión actual del kit.

4.2.1. Modificación en *MLAgents.Academy.cs*

Después de probar diferentes posibles soluciones, he optado por lo más sencillo: **llamar manualmente a la función de EnvironmentReset** desde mi código. Para poder hacer esto, sin embargo, ha sido necesario **modificar el nivel de acceso del método** desde el archivo de MLAgents *Academy.cs*.

A día de hoy, **no he encontrado referencias que indiquen una mejor manera de solucionar el problema**, así que sea o no una característica intencional del kit, recomendaría seguir este mismo curso de acción si se pretendiera hacer otra aplicación o videojuego con características similares.

4.3. Primer Entrenamiento: Toma de contacto

Después de solucionar los problemas encontrados y teniendo ya el entorno preparado para los entrenamientos, era momento de proceder con el **primer escenario**. Al ser el primero, se trata de el ejemplo más sencillo que puede montar, y su propósito es el de ofrecer una impresión honesta del funcionamiento de los agentes en fase preliminar.

En concreto, un agente de Lupus y otro de Red Nosed Hare aparecerán en el mapa número 1 (3). La intención es buscar el **comportamiento de caza** propio de Lupus y la huida de su presa, idealmente.

4.3.1. Estado de los agentes

Los agentes han sufrido cambios a lo largo del desarrollo desde sus fases preliminares, como se adelantaba en el capítulo 4.3.1. Para no hacer todas las fichas de los personajes de nuevo, simplemente se nombrarán los **cambios** respecto a la versión final, que se irán transformando de vuelta hasta llegar a las últimas fases.

Por ejemplo, Lupus:

- No utilizaba el **sensor de distancia**.

Y Red Nosed Hare:

- Recibía una **penalización de -0,1** por cada turno transcurrido.
- No recibía incentivo alguno por moverse.
- No recibía penalización por defenderse.

Además, ninguno de los dos utilizaba el sensor de **número de aliados vivos** (no era necesario en esta fase), y el **sensor de proximidad se encontraba en una versión temprana** que solo añadía valor a una dirección de movimiento (la versión final da valor a tres, lo que llamamos "dirección general") por cada enemigo encontrado.

4.3.2. Parámetros del Entrenamiento

Desde el proyecto se puede acceder al archivo *HEX.yaml* **con la configuración para los entrenamientos** en el formato definido por el kit de MLAgents. En esta sección se presentarán los **parámetros importantes** de este archivo y se justificarán los cambios competentes hechos en cada fase del entrenamiento.

- **Número de Comportamientos:** 2 (Lupus, RedNosedHare)
- **Hiper-parámetros de la Red Neuronal:**
  - **Algoritmo de Entrenamiento:** PPO
  - **Ratio de Aprendizaje:** 0.003.
  - **Beta:** 0.001
  - **Épsilon:** 0.2
  - **Lambda:** 0.99
- **Configuración de la Red Neuronal:**
  - **'Hidden Units':** 128
  - **'Número de capas':** 2
- **Máximo de Episodios:** 400.000

Para más información sobre el significado de los hiper-parámetros del algoritmo de PPO, visitar la página correspondiente en la documentación oficial de MLAgents [5]. Vale la pena adelantar, sin embargo, que muchos de ellos han permanecido invariables durante todo el proyecto, pues no ha parecido necesario cambiarlos.

4.3.3. Resultados Estadísticos

Mirar directamente los resultados estadísticos después del entrenamiento nos revelará datos importantes sobre cómo se ha desarrollado el proceso de aprendizaje.
Por un lado, si la **Entropía** desciende de manera más o menos constante a lo largo del entrenamiento, sabremos que el algoritmo está convergiendo correctamente (si el descenso fuera demasiado brusco o inexistente, tendríamos que tomar medidas, como cambiar el hiper-parámetro *Beta* o volver a enfocar el entrenamiento). Por el otro, tenemos la función de **Recompensa Acumulada**, que preferiblemente debería crecer hasta estabilizarse –lo que indica que el agente ha llegado a un máximo.
Los resultados estadísticos fueron positivos, con una **recompensa**

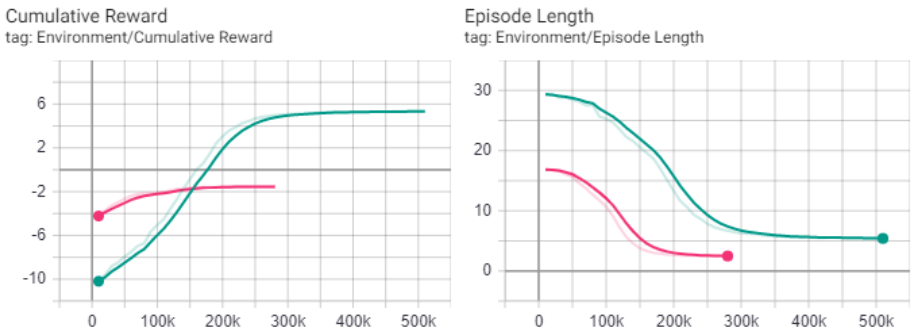

Figura 14: Función de Recompensa Acumulada y Duración Media del Episodio (Lupus en Verde, Red Nosed Hare en Rosa)

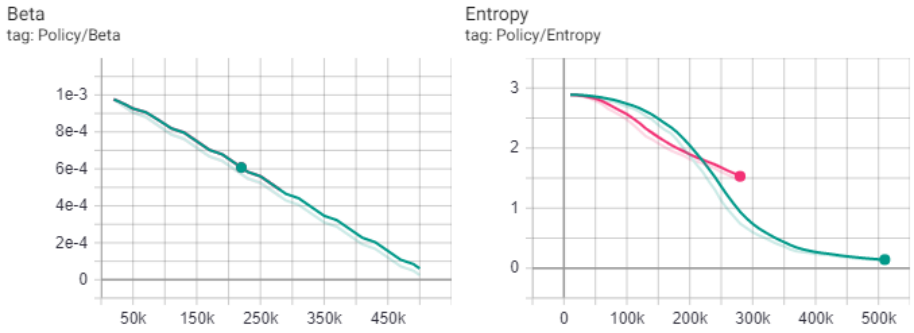

Figura 15: Entropía y factor Beta (Lupus en Verde, Red Nosed Hare en Rosa)

media máxima de 5.333 en Lupus y de -1.545 en Red Nosed Hare.

Estos resultados son esperables en un escenario en el que el agente Lupus haya ganado de manera consistente –que es el resultado deseado– pero era muy pronto para considerar el entrenamiento un éxito rotundo. Es importante juzgar los resultados visibles primero para **asegurarse de que las conductas son las apropiadas**, pues los agentes podrían haber aprendido algo diferente a lo planeado.

4.3.4. Resultados Visibles

Los resultados visibles revelan una buena y una mala noticia:

- La buena noticia era que **Lupus parecía comportarse de acuerdo al diseño previo**. Tras el entrenamiento, perseguía a su presa hasta lograr eliminarla.
- La mala noticia era que **Red Nosed Hare, sin embargo, había decidido no moverse en absoluto**.

Lupus parecía funcionar bien, en general, pero lo cierto es que su movimiento a veces se volvía errático e inconsistente, haciendo paradas innecesarias o intentando moverse hacia obstáculos, por lo que aún había cosas que mejorar.
Red Nosed Hare, por otro lado, había determinado que **no le compensaba intentar moverse** si iba a acabar muerto de todas formas. Las recompensas estaban probablemente mal niveladas: era preciso hacer un par de cambios.

4.4. Ajuste de Función de Recompensa

Para tratar de cambiar la tendencia a no moverse de Red Nosed Hare, decidí hacer algunos cambios a su función de recompensa. El problema del movimiento de Lupus, sin embargo, lo intentaría solucionar más adelante por separado.

4.4.1. Estado de los agentes

Los cambios para los siguientes entrenamientos fueron simples. Principalmente, **se le añadió una recompensa fija de +0.1 a Red Nosed Hare por moverse correctamente** (a direcciones válidas) y también **una penalización de -0.1 por defenderse.** Con estos cambios, pretendía incentivar activamente el movimiento mientras que castigaba la acción que había estado tomando antes por defecto.

4.4.2. Parámetros del Entrenamiento

Sabiendo que el entrenamiento anterior parecía converger sobre los 250.000-300.000 episodios, tenía sentido mantener el máximo de episodios a entrenar en 400.000. El resto de parámetros también seguirían igual.

4.4.3. Resultados Estadísticos y Resultados Visibles

Los parámetros estadísticos revelan que, probablemente, los cambios no habían sido lo suficientemente drásticos.
Como podemos observar en la figura 16, mientras que los cambios en la función de recompensa parecen notarse, –los valores en gris corresponden con los del entrenamiento más reciente– la duración media del episodio no varía en absoluto. Por las características de nuestro escenario inicial, esto puede ser un indicador claro de que **el curso de acción tampoco está cambiando**.

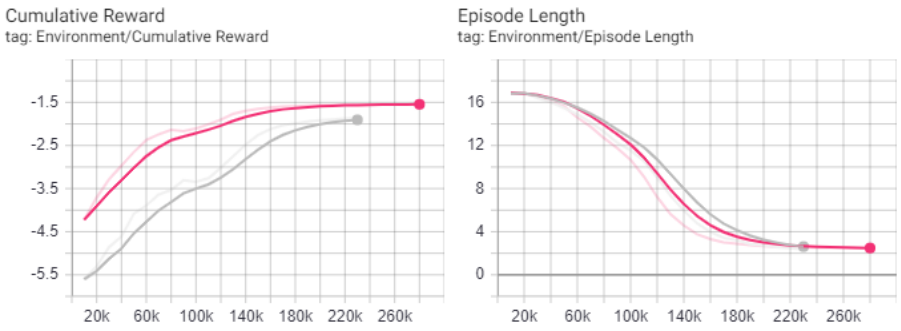

Figura 16: Comparación de los gráficos de Recompensa Acumulada entre la fase actual y la anterior para el agente Red Nosed Hare

Contrastando con los nuevos modelos de inferencia generados, este era exactamente el caso. De hecho, Red Nosed Hare ahora le facilitaba el trabajo a su depredador **desplazándose de vez en cuando hacia él**. En definitiva, **los cambios no habían resultado efectivos**.

4.5. Pruebas con los Parámetros de la Red Neuronal

En este tipo de situaciones, el curso de acción no es trivial. Antes de seguir haciendo pruebas con Red Nosed Hare, sin embargo, decidí intentar mejorar el comportamiento del agente que ya presentaba un mejor patrón de conducta.
Sin embargo, no quería abordar la solución que tenía pensada para intentar mejorar el movimiento antes de hacer algunas **pruebas con los parámetros de entrenamiento** de la Red Neuronal. Concretamente, pensaba modificar el valor **'hidden units'** o 'nodos ocultos'.

Este valor es un indicador de la **forma** de la red neuronal y, ajustado correctamente, puede suponer una mejora significativa en la velocidad de entrenamiento y en los resultados obtenidos. En este instante, el número de nodos ocultos era 128: ¿qué pasaría si aumentaba o disminuía ese número?
La respuesta corta es que, para este caso, **no suponía un cambio significativo**.

Si nos ceñimos a los resultados estadísticos (fig 17), comparando entre diferentes entrenamientos –el original con **128 'hidden units'**, un segundo con **256** y un **tercero con 64**– podemos observar que la diferencia parece ser mínima. Si acaso, la línea verde (el original) parece converger ligeramente más rápido.

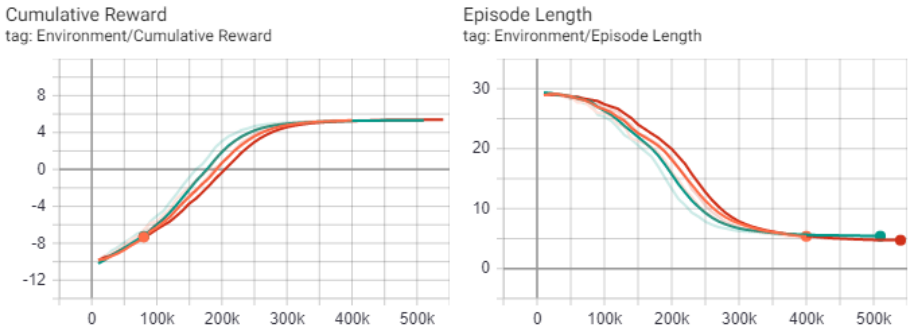

Figura 17: Comparación de Gráficos con distintos números de 'hidden units'

A fin de cuentas, los cambios no parecían ser lo suficientemente grandes como para justificar pasar a otra configuración de red neuronal, así que fueron desestimados cara a las fases siguientes.

4.6. Curriculum Learning para mejorar el Movimiento

Algo que ya había considerado es que, al entrenar a los agentes en un escenario sin obstáculos, **es posible que no supieran que hacer** si, de repente, se les soltaba en un entorno que sí los tuviera. En otras palabras: *el entrenamiento recibido hasta ahora no era extrapolable a cualquier tipo de escenario.*

4.6.1. Curriculum

Para solucionar esto, tenía preparada una división lógica en 'grados de dificultad' para los mapas diseñados. Mi criterio para establecer esta división es, básicamente, el número de obstáculos o potenciales 'callejones sin salida' que un agente pudiera encontrar. Estableciendo esta división podría añadir una **variable de entorno (map_config)** a *HEX.yaml*, que indicara qué fracción del set de mapas a utilizar. Se trata de un número entero que toma valores entre 0 y 6 según la fase del entrenamiento.

```
map_config:
curriculum:
  - name: SimplerMaps
    completion_criteria:
      measure: reward
      behavior: Lupus
      signal_smoothing: true
      min_lesson_length: 100
      threshold: 0.0
    value: 3.0
  - name: HarderMaps # start of the second lesson
    completion_criteria:
    value: 6.0
```

El valor (value) de la variable se corresponde con el **índice máximo de mapa que puede cargar**, y se carga un mapa elegido aleatoriamente de entre el total disponible cada episodio. Con esto y con spawns semi-randomizados de los NPC, se garantiza suficiente variedad como para que el entrenamiento sea aplicable a cualquiera de los mapas diseñados en una variedad de situaciones.

4.6.2. Estado de los agentes

En esta fase, además, se incorporaron algunos cambios clave en los agentes:

Por un lado, los **sensores de distancia hasta la presa más cercana** de Lupus, además de un poco de **feedback positivo** inversamente proporcional a la distancia que le separa de dicha presa.
Adicionalmente, se **actualizó el sensor de proximidad**, reflejando ya el estado definitivo tal y como aparece en la sección de algoritmos al final de la memoria.

Estos cambios pretendían ayudar a solventar los problemas en el movimiento de ambos agentes. Se esperaba que, con la ahora mayor variedad de entornos de entrenamiento, el movimiento acabase siendo más correcto en un mayor número de situaciones.

4.6.3. Resultados Estadísticos

Los resultados estadísticos aportan diferentes datos:

- Se puede observar el punto en el que se entra en la **siguiente fase** del curriculum learning –se hace aparente en el pequeño bache en el progreso de Lupus sobre la marca de 175.000 episodios. El velocidad de adaptación es sorprendentemente alta y la función de recompensa acumulada llega aproximadamente al mismo valor anterior.
- El las gráficas de Red Nosed Hare siguen sin sufrir cambios drásticos, lo que no es buena señal para esta fase de entrenamiento.
- La **duración** de los episodios se ve afectada de manera importante por el cambio de entornos, como es lógico, pues desplazarse por unos mapas es relativamente más costoso que por otros.

Cabe añadir que, no sabiendo cuánto tardarían los agentes en acostumbrarse a los nuevos entornos, se aumentó previamente el número

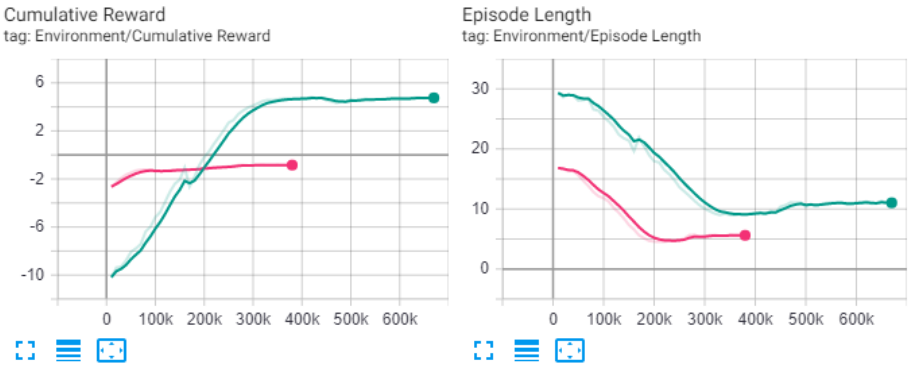

Figura 18: Resultados Estadísticos con Curriculum Learning

máximo de episodios a 1 millón, pero al observar que la función no parecía moverse pasado un punto, se paró sobre las 700.000 iteraciones de Lupus.

4.6.4. Resultados visibles

Usando los modelos de inferencia generados durante el entrenamiento, podemos observar de primera mano lo que ya revelaba la estadística: el **comportamiento de Red Nosed Hare no había mejorado con los últimos cambios introducidos**.
Por otro lado, el movimiento de Lupus había mejorado bastante en los mapas con obstáculos, aunque **ahora parecía tener problemas atacando a su presa** y priorizaba acecharla en un número alarmante de ocasiones.

En general, había progresos con Lupus, pero parecía que la **recompensa añadida al movimiento** estaba siendo contraproducente.

4.7. Curriculum Learning para Diferentes Poblaciones

En esta última fase se han terminado de realizar los cambios que restaban en los agentes. El más importante, la **recompensa pasiva por supervivencia** de +0.1 a Red Nosed Hare. Y es que, tras probar sin éxito cientas de modificaciones a su función de recompensa, no había reparado en lo que era el quid de la cuestión: si el agente recibe penalizaciones por el simple hecho de existir, lo más natural es que quiera terminar el juego lo más rápido posible –en este caso, facilitando la tarea de su cazador.
Además, viendo que la **recompensa por movimiento que recibía Lupus** podía estar causando problemas, se ha acabado eliminando por completo, esperando que este cambio, unidos a la mejora de su movimiento anterior, lo acercara más al estado de diseño.

4.7.1. Curriculum

Cuando se añadían diferentes números de agentes a cada facción, los resultados no eran del todo malos, pero quería probar si era posible entrenar la colaboración y la consistencia de los agentes en estos nuevos entornos.
La intención detrás de esta última serie de experimentos era conseguir adaptar las redes neuronales **ya entrenadas** tras los últimos cambios antes mencionados a **distintos ecosistemas posibles**: presas en superioridad numérica, grupos de cazadores... situaciones en las que la colaboración puede cambiar el transcurso de la batalla.

4.7.2. Parámetros del Entrenamiento

A las variables de entorno he añadido dos parámetros: *fact1_config* [1-2 unidades] y *fact2_config* [1-2-3 unidades]. Son números enteros que controlan el número de agentes inicializados en el mapa al comienzo de cada episodio y son los que cambiaré en cada prueba.

4.7.3. Resultados

En primer lugar, cabe resaltar que los resultados para el escenario base han mejorado considerablemente, principalmente por el nuevo comportamiento de Red Nosed Hare, mucho más acorde con su propósito de diseño. Los resultados estadísticos de esta primera iteración (figura 19) muestran como Red Nosed Hare progresa más allá de sus límites anteriores.
A partir de este momento, Red Nosed Hare había aprendido **a huir**

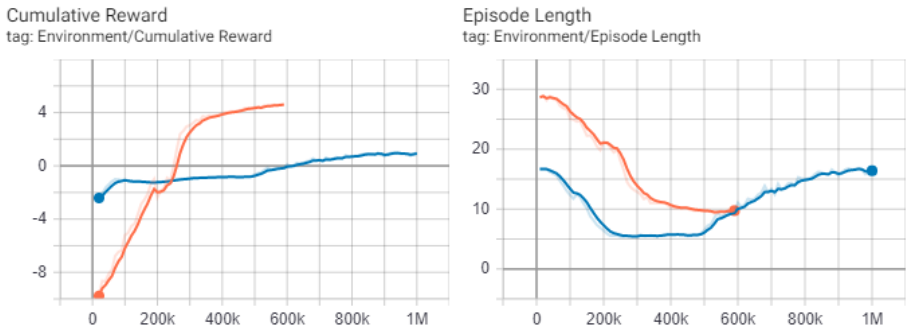

Figura 19: Funciones de Recompensa Acumulada para Lupus (naranja) y Red Nosed Hare (azul). A partir de las 600K iteraciones, Red Nosed Hare se enfrenta al modelo de inferencia generado para Lupus

de su depredador, alargando su vida y, por ende, aumentando la duración media del episodio junto con el valor de su función de recompensa.

Para los siguientes escenarios, inicialicé el proceso de aprendizaje a partir de los modelos de inferencia anteriores. Sobre el papel, esto podía conseguir adaptar comportamientos ya entrenados a diferentes escenarios, pero la realidad resultó ser bastante distinta.
Los agentes que pasaron por este proceso acabaron desarrollando conductas erráticas y poco útiles pero, *¿por qué?*

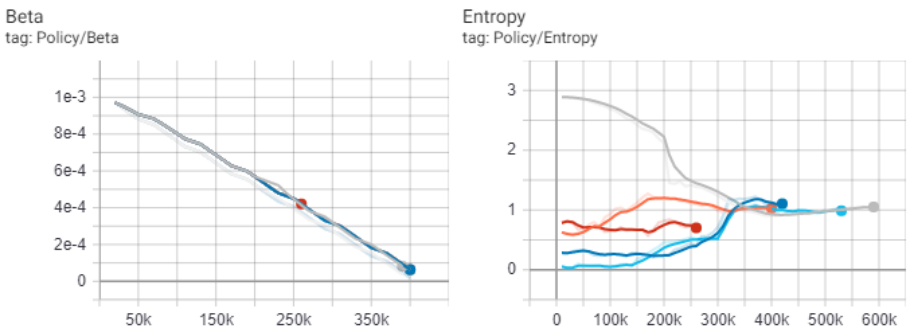

Figura 20: Entropía y Factor Beta para las diferentes iteraciones de ecosistemas inicializados desde el entrenamiento base.

La respuesta la da, en parte, el gráfico de la Entropía a lo largo del entrenamiento (figura 20) de las diferentes iteraciones. Los ejemplos apuntan a que los entrenamientos no han progresado bien y han degenerado en conductas más **aleatorias** sin llegar a converger.

Los resultados de entrenar estos ecosistemas por separado tampoco eran reveladores: principalmente **Lupus parece tener problemas para distinguir miembros de su propia facción**, lo que le lleva a bloquearse. Con los gráficos correspondientes (Lupus en azul, figura 21): llegados a cierto punto, **Lupus deja de intentar cazar a sus presas.**

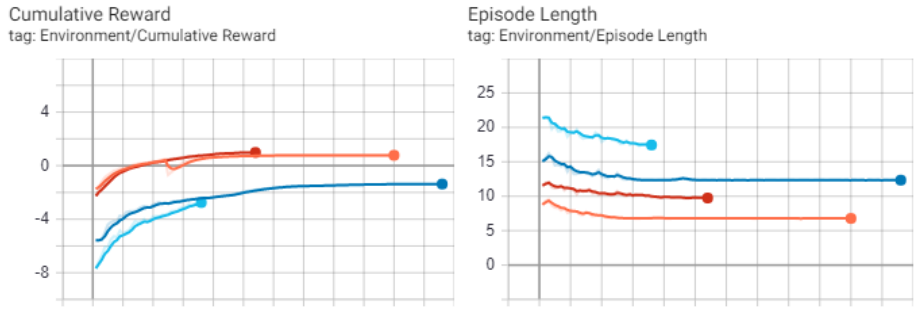

Figura 21: Resultados estadísticos de las diferentes versiones de Lupus y Red Nosed Hare, entrenadas desde cero en diferentes ecosistemas.

Al margen de estos ejemplos, existe uno que sale de la norma. El entrenamiento en el ecosistema con dos presas y un cazador, inicializado a partir del entorno base, desemboca en comportamientos lógicos –aunque no perfectos.

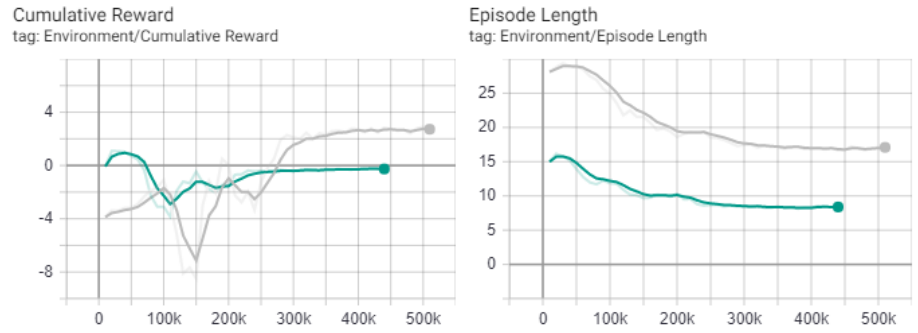

Figura 22: Funciones de Recompensa Acumulada para Escenario con 2 Red Nosed Hare (verde) y 1 Lupus (gris). Inicializado a partir del modelo base.

Los agentes conservan las intenciones deseadas, pero el cazador puede llegar a preferir moverse primero hacia la presa más alejada, lo que a efectos prácticos ralentiza su victoria. El agente Lupus, en su estado actual, **tiene también problemas lidiando con más de una presa**, lo que puede deberse a *dificultades para convertir las señales que recibe de los sensores en información útil.*

5. Conclusiones y Líneas Futuras

Los resultados obtenidos hasta el momento son positivos.
Por un lado, se ha conseguido un **videojuego robusto** con un entorno bien definido donde pueden coexistir agentes inteligentes bajo una serie de reglas que regulan su conducta.
Por el otro, tenemos resultados de entrenamientos que apuntan a diversas conclusiones:

- Entrenar agentes con técnicas de Reinforcement Learning para los escenarios propuestos da **buenos resultados, comparables a las heurísticas propuestas** y, en ocasiones, resultando en patrones de acción más orgánicos y fácilmente adaptables.
- El movimiento con obstáculos en el terreno es difícil de entender completamente para los agentes. Sin embargo, el uso de técnicas de Curriculum Learning para **aumentar progresivamente la dificultad de los mapas** mejora considerablemente el desempeño.
- El uso de técnicas de Curriculum Learning para adaptar el comportamiento de los agentes a diferentes números de aliados/enemigos en el entorno da resultados prometedores, pero mejorables para algunos agentes. Probablemente este problema se podría abordar a través de algún **mecanismo que permita a los agentes comunicar sus intenciones**, o de **observaciones más precisas**.
- Recompensar correctamente los cursos de acción deseados cara al entrenamiento es de un valor primordial para este tipo de trabajos. Entender el propósito de los agentes y organizar sus funciones de recompensa acorde es un aspecto que cambia completamente los resultados de un entrenamiento.

A modo de cierre, además, hay varios caminos que resultaría interesante tomar para ampliar el alcance del proyecto:

- El primer paso lógico sería **dar vida a todos los agentes diseñados** que no han recibido entrenamientos, como *Matriarch Hare* o *Lupus Alpha*. Como con los demás agentes, habría que observar primero como funcionan los diseños previos en los entornos actuales, para luego adaptarlos según fuese necesario.
- También sería conveniente terminar de adaptar a los agentes para que desarrollen comportamientos variables en función de

las características de su ecosistema, como se pretendía con los últimos experimentos.

- Por otro lado, también me gustaría aumentar el número de mapas, añadiendo diseños **más desafiantes para los agentes** que terminen de refinar sus patrones de movimiento.
- Otro aspecto sobre el que valdría la pena indagar sería el comportamiento de los agentes si **aumentáramos radicalmente el número y la calidad de las observaciones**: observaciones específicas para cada una de las demás unidades en juego que pudieran impulsarles a desarrollar **conductas más elaboradas** o incluso **estrategias colaborativas**.
- Por supuesto, sería importante **integrar finalmente los agentes en el bucle de juego real**, incluyendo al jugador en el mismo. A partir de aquí habría que evaluar la experiencia de juego resultante y hacer una comparativa estricta con los diseños de comportamiento heurísticos.
- Y por último, me gustaría **pulir los aspectos artísticos y estilísticos del juego**: mejorar su atractivo, añadir texturas, sonido y animaciones a la batalla y dar realmente 'vida' a los personajes.

En definitiva, podemos concluir que se han obtenido resultados prometedores que demuestran la factibilidad del uso de técnicas de Deep Learning para entrenar a los NPC en juegos de estrategia por turnos. Mientras que no es seguro que esto vaya a convertirse en la norma para el futuro de este género, siempre quedará como opción y como una alternativa más *elegante* a enfoques clásicos como son la elaboración de enormes y laboriosos árboles de decisión que, por otro lado, dan lugar a comportamientos mucho menos flexibles y mucho más predecibles.

6. Summary and Conclusions

Obtained results so far are certainly positive.
On the one hand, we have a **robust game design** with well-defined environments where intelligent agents can coexist, all under a series of rules and restrictions that limit their actions.
On the other hand, we have the training results, which lead to different conclusions:

- Training agents with Reinforcement Learning techniques for the proposed environments has given **great results, comparable to their counterpart heuristic approaches** and, ocasionally, it has resulted in even more organic and easily adaptable behaviours.
- Movement with obstacles around the map has proven to be a difficult task for the agents to figure out. However, the use of Curriculum Learning techniques to **progressively increas the maps' difficulty** has shown to improve results considerably.
- The use of Curriculum Learning techniques to adapt agents' behaviors to different numbers of enemies/allies has given promising results, but these could also be improved for some agents. This is an issue that could be probably solved through some sort of **mechanism or system with which the agents can communicate their intentions to each other**, or with a better sensor quality overall.
- The importance of encouraging the correct courses of action for the agents has been proven critical once again. Understanding the purpose of the different agents and tweaking their reward functions accordingly is a task that may heavily influence the training results.

To wrap up, there are also various future courses of action that would be interesting to take with this project to expand its reach:

- First logical step would be to **implement all of the designed agents** that have yet not been part of the training phase, like *Matriarch Hare* or *Lupus Alpha*. Like with all of the other agents, it would be advised to first test the preliminary designs on current working scenarios to then adapt and improve them as necessary.

- In addition to this, I expect to completely adapt the agents to help them reach flexible behaviours for different sorts of environments as it was intended in the last group of experiments.
- I would also like to increase the number of maps, providing the agents with **more defying designs** that help to completely refine their movement strategies.
- Another aspect that seems worth the test is the training results for **radically different** designs, with **a higher number and quality of the observations:** specific data for each and every unit in the game directly fed to the agent could maybe push them to **new heights** and/or **more elaborate or even collaborative strategies**.
- Of course, **integrating the agents on the game loop** alongside the player would be important. From there, we could evaluate the *gaming experience* and the *feel* of the game when faced with heuristic agents versus reinforcement learning agents.
- And last but not least, it would be nice to **improve the more artistic aspects of the game:** attractive, textures, sound, battle animations and all of the elements that truly make games feel 'alive'.

In conclusion, some very promising results have been gathered, where the feasibility and usage of Deep Learning techniques to train NPCs for turn-based, strategy games truly show. And while I am not certain of these becoming the new norm, or the future for the genre, it is clear that they will always be a more **elegant** alternative to heuristic approaches, like the construction of enormous and laborious decision trees which, on the other hand, often result in much less flexible and more predictable behaviours.

Bibliografía

[1] Universidad de La Laguna, Grado de Ingeniería Informática
https://www.ull.es/grados/ingenieria-informatica/

[2] OpenAI
https://openai.com/

[3] HEX Battlegame - Github Project Page
https://github.com/alu0101045870/HexBattleGame

[4] Final Fantasy X Stat Mechanics (by SinirothX and SClemmons)
https://gamefaqs.gamespot.com/ps2/197344-final-fantasy-x/faqs/31381

[5] Hexagonal Grids - From Red Blob Games
https://www.redblobgames.com/grids/hexagons/

[6] Unity ML-Agents Docs: Training with Proximal Policy
https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md

Métodos y Código Fuente Referenciado

Bucle de batalla

```
/*****************************************************************************
*
* Fichero BattleMap.cs
*
*****************************************************************************
*
* AUTOR Fernando González Petit
*
* FECHA 1-09-2020
*
* DESCRIPCION El "bucle de juego" dispuesto en el modo entenamiento. Se puede observar
*   la secuencia de eventos explicada en el capítulo 2 con un mayor lujo de detalle
*****************************************************************************/
IEnumerator TrainingLoop()
{
    bool stopCondition = false;
    int index;
    while (!stopCondition)          // training loop will set stopCondition to false
    {                               // when the battle loop is done to start over
        index = caroussel.NextTurnOwner();               // current turn holder set
        caroussel.actionInfo.TurnOwner = battleUnits_[index];

        for (int j = 0; j < battleUnits_[index].GetStatValueByName("ACT"); j++)
        {
            battleUnits_[index].RequestAct();           // Requests agent Action
            Academy.Instance.EnvironmentStep();
            yield return new WaitUntil(() => battleUnits_[index].ActionOver);
            yield return new WaitForSeconds(1f);        // Ensure animations are over

            // Training Over condition: MaxStep Reached
            if (stopCondition =
                 (battleUnits_[index].MaxStep < battleUnits_[index].StepCount)) break;

            if (caroussel.CheckCarousselTriggerEvents())
            {
                // Battle Over condition: Faction Supremacy
                if (stopCondition = FactionSupremacy(out winnerFactions)) break;
            }
        }
        // After-Turn events (poison, etc) should be triggered here <======
        battleUnits_[index].PostTurnEvents();

        if (!stopCondition)
        {
            battleUnits_[index].SynthethiseSkillRanks();    // Decides turn skill rank
            caroussel.PassTurn();
        }
        else
        {
            stopCondition = false;
            RewardWinners(winnerFactions);
            EpisodeReset();
        }

        caroussel.actionInfo.Reset();
    }
}
```

Cálculo de turnos

```
/*****************************************************************************
*
* Fichero .h
*
*****************************************************************************
*
* AUTORES
*
*
* FECHA
*
*
* DESCRIPCION
*
*
*****************************************************************************/
/// <summary>
/// Gives the order of passing turns to the caroussel, which does
/// all necessary calculations depending on game events
/// </summary>
public void PassTurn()
{
    // Decrease status effects of current turn owner
    actionInfo.TurnOwner.DecreaseStatusCounters();
    UpdateCounterValues();

    // Dequeue the first turn entry
    entries_.Dequeue();

    // second, actually dequeue from the scene gameObject
    Destroy(gameObject.transform.GetChild(0).GetChild(0).gameObject);

    // if => haste has been applied
    //       skill rank has changed (not 3)
    //       a character has died
    // then: re-calculate the full queue
    if (CheckCarousselTriggerEvents())
        PreCalculateTurns();

    // else: calculate and assign next turn
    else SetNextTurn(CalculateNextTurn());

    // UI refreshes on Update
}

/// <summary>
/// Calculates and assigns forward all 16 turns for the game
/// </summary>
public void PreCalculateTurns()
{
    int currentlyCaltulatedTurns = 0;

    // Clear previous turn queue
    ClearPreviousQueue();
    ResetDefaultTurns();
    ReSetForwardCounterValues();

    //
    // Calculate a "first round" of turns
    // For the current LastSkillRanks
```

```
    while (currentlyCaltulatedTurns < PRE_CALCULATED_TURNS)
    {
        SetNextTurn(CalculateNextTurn());
        currentlyCaltulatedTurns++;
    }

    // check flag is active => order is maintained in
    // forward SetTurn routine
}

private Pair<int, int> CalculateNextTurn()
{
    List<GameCharacter> battleUnits = battleMap.battleUnits_;
    int index = 0;
    bool turnFound = false;

    // only Active battleUnits are given turns
    while (!turnFound)
    {
        for (int i = 0; i < battleUnits.Count; i++)
        {
            if (battleUnits[i].IsActive && counterValues[i] <= 0 && !turnFound)
            {
                turnFound = true;
                index = i;
            }

            counterValues[i]--;
        }
    }

    if (NonDefaultTurnsToBeAssigned())
    {
        defaultTurnsToBeAssigned[index] = true;
        return new Pair<int, int>(index, battleUnits[index].LastSkillRank);
    }
    else
    {
        // Queue a default skillRank turn
        return new Pair<int, int>(index, 3);
    }
}
```

Sensor de Proximidad

```
/*****************************************************************************
*
* Fichero Enemy.cs
*
*****************************************************************************
*
* AUTOR: Fernando González Petit
*
* FECHA: 01-09-2020
*
* DESCRIPCIÓN: El Sensor de Proximidad provee al agente de un indicador del valor
*   potencial que otorgaría moverse en cada una de las seis direcciones posibles.
*
*****************************************************************************/

/// <summary>
/// This method is one of the Sensors that are used to provide information about
/// the battle grid to the agents. The Proximity Sensor gives an indicator of
/// the potential 'value' that moving towards each available direction holds.
/// </summary>
/// <returns>A List of Floats [size 6] representing
/// the value ratios for each direction</returns>
protected virtual List<float> ProximitySensor()
{
    List<float> proximitySensor = new List<float>(new float[]{ 0f, 0f, 0f, 0f, 0f, 0f});

    // Search for targets and predators in map
    List<GameCharacter> detectedObjectives = ObjectivesInSightSensor();
    List<GameCharacter> detectedPredators = PredatorsInSightSensor();

    List<int> generalDir;
    int dir;

    // Add some value to the general directions in which there are potential targets!
    for (int i = 0; i < detectedObjectives.Count; i++)
    {
        dir = HexCalculator.GeneralDirectionTowards(InGamePosition,
            detectedObjectives[i].InGamePosition);
        generalDir = HexCalculator.ForwardDir(dir);

        proximitySensor[generalDir[0]] += 1f;
        proximitySensor[generalDir[1]] += 0.5f;
        proximitySensor[generalDir[2]] += 0.5f;
    }

    // Add some value to the OPPOSSITE directions to which there are hungry predators!
    for (int i = 0; i < detectedPredators.Count; i++)
    {
        dir = HexCalculator.GeneralDirectionTowards(InGamePosition,
            detectedPredators[i].InGamePosition);
        generalDir = HexCalculator.OppositeDir(dir);

        proximitySensor[generalDir[0]] += 1f;
        proximitySensor[generalDir[1]] += 0.5f;
        proximitySensor[generalDir[2]] += 0.5f;
    }

    return proximitySensor;
}
```

Sensor de Adyacencia

```
/*****************************************************************************
*
* Fichero Enemy.cs
*
*****************************************************************************
*
* AUTOR: Fernando González Petit
*
* FECHA: 01-09-2020
*
* DESCRIPCIÓN: El Sensor de Adyacencia provee al agente de un indicador preciso con
*   información sobre las casillas que le son directamente adyacentes.
*****************************************************************************/

/// <summary>
/// This method is one of the Sensors that are used to provide information about
/// the battle grid to the agents. The Adjacency Sensor is meant to provide with
/// a precise indicator of the map tiles direcly adjacent to the agent
/// </summary>
/// <returns>A List of Floats [size 6] with different fixed values
/// for each direction. Each value holds a different meaning</returns>
protected virtual List<float> AdjacencySensor()
{
    List<float> adjacencySensor = new List<float>();
    HexTile currentTile = BattleMap_.mapTiles[InGamePosition];
    HexTile neighbor;

    for (int dir = 0; dir < 6; dir++)
    {
        // if there is a neighboring tile at dir
        if (currentTile.Neighbors.TryGetValue(dir, out neighbor))
        {
            if (neighbor.Occupied)
            {
                // if tile is occupied by an enemy or prey
                if (OccupierInTargetList(neighbor))
                {
                    adjacencySensor.Add(0.5f);      // target at dir
                }
                else if (OccupierInPredatorList(neighbor))
                {
                    adjacencySensor.Add(1f);
                }
                else
                {
                    adjacencySensor.Add(-0.5f);     // no target at dir, but not empty
                }
            }
            else
            {
                adjacencySensor.Add(0f);            // empty tile
            }
        }
        else
        {
            adjacencySensor.Add(-1f);               // obstacle tile
        }
    }
    return adjacencySensor;
}
```