Trabajo Final de Grado:

Desarrollo de un bot para un juego de lucha mediante aprendizaje por refuerzo

Memoria

David Balaghi Buil

<u>Director</u>: Javier Béjar Alonso (Ciencias de la computación)
23 de Octubre de 2018
Especialidad *Computació*

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) - BarcelonaTech
FACULTAT D’INFORMÀTICA DE BARCELONA (FIB)

Resum

Aquest projecte representa el final del Grau en Enginyeria Informàtica de la FIB. És per això que un dels seus objectius és posar en pràctica bona part dels coneixements estudiats al grau, particularment els de l'especialitat de Computació.

El projecte tracta sobre el desenvolupament d'un videojoc de lluita 2D (The Fighting Game) i de la implementació d'un bot que anirà aprenent a jugar amb el propi jugador mitjançant l'ús de tècniques d'aprenentatge per reforç. El videojoc es desenvoluparà utilitzant el motor Unity, tot plegat amb la seva API de *learning agents.*

Aquest document està dividit en diverses parts: Una introducció per a contextualitzar el problema, el disseny del videojoc i la seva IA, la implementació en Unity, una anàlisi dels resultats obtinguts, acabant amb un apartat de gestió de projecte i conclusions.

Resumen

Este proyecto representa el final del grado en Ingeniería Informática de la FIB. Por ello, uno de sus objetivos es poner en práctica buena parte de los conocimientos estudiados en el grado, y particularmente en la especialidad de Computación.

El proyecto trata sobre el desarrollo de un videojuego de lucha en 2D (The Fighting Game) y de la implementación de un bot que irá aprendiendo a jugar junto al jugador mediante el uso de técnicas de aprendizaje por refuerzo. El videojuego se desarrollará utilizando el motor Unity, junto con su API de *learning agents*.

Este documento está dividido en varias partes: una introducción para contextualizar el problema, el diseño del videojuego y la IA, su implementación en Unity, un análisis de los resultados obtenidos, acabando con un apartado de gestión del proyecto y conclusiones.

Abstract

This project represents the end of the Computer Engineering Degree at FIB. That's why one of its objectives is to bring out the skills that were acquired all these years, especially those learned at the Computer Science branch.

The project's objective is to develop a 2D fighting game (The Fighting Game) and to implement a bot that will learn to play while it fights the player by using Reinforced Learning techniques. The game will be developed using the Unity Engine, as well as its *learning agents* API.

This document is divided into parts: an introduction for the problem's scope, the game and AI design, their implementation in Unity, a results analysis, and finally a section to discuss the project management and conclusions.

Índice

1. Introducción **5**
1.1 Vocabulario 5
1.2 Contexto 6
1.2.1 Descripción general 6
1.2.1.1 Objetivos 7
1.2.2 Actores implicados 8
1.2.3 Influencias 9
1.2.4 Alcance del proyecto 10
1.2.4.1 Posibles obstáculos 11
1.2.5 Alcance de la memoria 12
1.3 Estado del arte 12
1.3.1 Videojuegos e inteligencia artificial 12
1.3.2 Desarrollo de videojuegos 14
1.3.3 Aprendizaje por refuerzo 15
1.3.4 Leyes y regulaciones 15
1.3.4.1 PEGI 15
1.3.4.2 ESRB 16
1.3.4.3 Otros 16
1.4 Estructura del documento 17
2. Descripción del videojuego **18**
2.1 Interfaz 18
2.2 Jugador 19
2.2.1 Jugabilidad 19
2.2.1.1 Movimiento 19
2.2.1.2 Acciones 20
2.2.2 Controles 21
2.2.3 Apartado artístico 22
2.3 Mecánicas del juego 23
2.4 Modos 25
2.4.1 Modo normal 25
2.4.2 Modo entrenamiento 26
2.5 Diseño de la IA 26
2.5.1 Q-Learning 27

2.5.2 La variante que usaremos: PPO 28
2.5.3 Estrategia 1 29
2.5.4 Estrategia 2 30
2.5.5 Estrategia 3 31
3. Implementación 32
3.1 Escena 32
3.2 Objeto ‘Player’ 33
3.2.1 Sprite 33
3.2.2 Animaciones 34
3.2.3 Físicas 44
3.2.4 Script 45
3.2.4.1 Movimiento 46
3.2.4.2 Acciones y timers 50
3.2.5 Detalles finales 53
3.3 Otros objetos 56
3.3.1 Cámara y limitadores de escenario 56
3.3.2 Fondo y UI 57
3.4 IA del bot 58
3.4.1 Objetos de la API 58
3.4.2 Modos del Brain 59
3.4.3 El agente 59
3.4.3.1 Observaciones 60
3.4.3.2 Acciones y recompensas 61
3.4.3.3 Adaptando el Input del personaje al agente 63
4. Descripción de resultados 66
4.1 Estrategia 1 67
4.1.1 Tabla de recompensas 67
4.1.2 Análisis 67
4.2 Estrategia 2 69
4.2.1 Tabla de recompensas 69
4.2.2 Análisis 69
4.3 Estrategia 3 71
4.3.1 Tabla de recompensas 71
4.3.2 Análisis 71
5. Gestión del proyecto 73
5.1 Metodología 73
5.1.1 Seguimiento 73
5.1.2 Herramientas de desarrollo 73
5.2 Planificación 74
5.2.1 Hito inicial 75
5.2.2 Análisis y diseño 75

5.2.3 Descripción de las tareas 75
5.2.4 Hito final 76
5.2.5 Duración aproximada 76
5.2.6 Valoración de alternativas 77
5.2.7 Diagrama de Gantt 78
5.2.8 Recursos para el desarrollo del proyecto 79
5.2.8.1 Hardware 79
5.2.8.2 Software 79
5.3 Gestión económica 79
5.3.1 Recursos humanos 79
5.3.2 Hardware 80
5.3.3 Software 80
5.3.4 Licencias 81
5.3.5 Gastos indirectos 81
5.3.6 Presupuesto total 81
5.4 Sostenibilidad 82
5.4.1 Dimensión ambiental 82
5.4.2 Dimensión social 82
5.4.3 Matriz de sostenibilidad 83
6. Conclusiones 84
6.1 Posibles ampliaciones 84
Bibliografía 86

1. Introducción

1.1 Vocabulario

En este primer apartado, introduciremos brevemente algunos conceptos que pueden ser útiles para la lectura de esta memoria. Se trata principalmente de conceptos relacionados con desarrollo de videojuegos e IA.

Fighting game

Un *fighting game*, generalmente, se le denomina a aquellos juegos que tratan de dos o más personajes que se enfrentan en combate, ya sea usando artes marciales o habilidades especiales de todo tipo. Es uno de los géneros de videojuego más antiguo que existe, ya que fueron de los primeros tipos de juegos que se presentaban en las máquinas *arcade* de los salones recreativos. Se caracteriza por su carácter competitivo y su gran facilidad de permitir a varios jugadores jugar unos contra otros.

Bot

Un bot es un programa que usa alguna técnica de inteligencia artificial para actuar de forma autónoma en un entorno. En el caso de los videojuegos, los bots son jugadores que funcionan mediante una IA, con el objetivo de interactuar con las personas que juegan, ya sea enfrentándose a ellas o ayudándolas. Todo depende del diseño de cada videojuego.

Aprendizaje por refuerzo

El aprendizaje por refuerzo es una técnica de IA que se basa en, dado uno o más agentes, un conjunto de estados, y acciones que llevan de un estado a otro, se computa una lista de pesos para cada acción con el objetivo de maximizar la ganancia final. La computación de pesos se genera a partir de recompensas y penalizaciones que el entorno envía al agente en cada estado, de manera que poco a poco va aprendiendo qué acciones dan mejor resultado, de forma autónoma.

1.2 Contexto

Para el desarrollo de este proyecto, hemos tenido que centrarnos en los dos contextos principales del mismo: El desarrollo del videojuego, y la implementación de la Inteligencia artificial (IA) del bot.

El primero es algo muy trabajado en el mundo a día de hoy, al igual que el segundo; en particular las técnicas de aprendizaje por refuerzo, que se pueden ver no solo en juegos, pero en otros tipos de sistemas inteligentes. Pero la mezcla de ambos contextos es lo que le da una nueva y amplia visión al desarrollo del proyecto. A pesar de que hoy en día prácticamente todos los videojuegos cuentan con algún tipo de IA, el caso particular de una IA que aprenda y mejore es bastante menos común.

En particular, la IA de los videojuegos suele usarse para generarle algún tipo de desafío al jugador. Por ejemplo, es el caso de los enemigos o rivales a los que debes enfrentarte en un juego competitivo, o simplemente los NPC[1] enemigos en un juego de acción o aventuras, que se enfrentan al jugador en su camino. Este es el caso que se tratará en este proyecto.

Pero no necesariamente deben oponerse al jugador. Se puede desarrollar una IA para personajes que cooperen con el jugador a lo largo del juego, como es el caso de *Shooters*[2] o *RPG*[3], entre otros, en los cuales tienes compañeros controlados por la máquina que te ayudan en tu partida. En algunas ocasiones puedes incluso dar órdenes o programar alguna parte de su comportamiento, lo cual los convertiría en NPC semi-controlados por el jugador.

1.2.1 Descripción general

A lo largo de la historia de los videojuegos, numerosos géneros y estilos de juego se han ido desarrollando, con el objetivo de entretener y divertir a los jugadores.

---

[1] Non Playable Characters. Personajes o entidades del juego que tienen algún tipo de comportamiento no controlado por el jugador.

[2] Shooters es el género de videojuegos en que el personaje usa armas de fuego o similares. Suelen ser en primera persona, aunque algunos se juegan en tercera persona.

[3] Role Playing Games. Juegos de rol, donde el personaje principal suele tener algún tipo de progresión a lo largo del juego, y sus habilidades se determinan a partir de atributos del propio personaje.

Originalmente los videojuegos solían enfrentar a los usuarios entre ellos en juegos competitivos, ya que las inteligencias artificiales implementadas eran más bien limitadas y sencillas.

A lo largo de los años se ha ido trabajando en técnicas de IA para hacer los juegos más entretenidos de jugar, en términos de *player vs computer* (el jugador contra la máquina), para intentar que se pudiera jugar a videojuegos competitivos en soledad, o en modos cooperativos (varios jugadores contra la máquina).

El problema que ha traído consigo el desarrollo de dichas inteligencias artificiales es, principalmente, que su programación era "estática", o dicho de otra manera, están preparadas para jugar de una forma predefinida, y no cambian ni mejoran a medida que compiten contra el jugador. Esto conlleva que, dado que los jugadores mejoran con el tiempo, la máquina acaba quedándose atrás, y competir contra la IA del juego se hace aburrido o repetitivo.

Es por esta razón que en este proyecto se va a desarrollar un estilo de IA más complejo. Se trata de un bot[4] que va aprendiendo a medida que juega, como si de una persona se tratase, de manera que siempre presente un desafío para el usuario, haciendo que el juego siempre se mantenga interesante, sin necesidad de jugar y competir contra otras personas.

Para ello, se desarrollará un videojuego competitivo de lucha[5], con todas las funcionalidades básicas de un juego de este género, sobre el cual se implementará el bot que se enfrentará al jugador.

1.2.1.1 Objetivos

Una vez conocido el problema sobre el que queremos trabajar, es necesario plantear una lista de objetivos a alcanzar con el desarrollo del proyecto.

Dichos objetivos se podrían englobar en los siguientes puntos:

---

[4] Un jugador artificial, es decir, controlado por una inteligencia artificial, en vez de por una persona física.

[5] Un juego de lucha es aquel que enfrenta a dos o más personajes controlados o no por un jugador, en un escenario, y acaba cuando determinadas condiciones se dan (un personaje se queda sin vida, se sale del escenario, etc).

- Realización de un estudio y análisis de las distintas técnicas de aprendizaje por refuerzo y establecer las opciones más adecuadas para la implementación del bot.
- Desarrollo del propio videojuego, y todo lo que derive de ello (diseño, implementación, testing, etc).
- Implementación del bot sobre el videojuego desarrollado.
- Comprobación y manipulación de los parámetros del bot para asegurarse de que cumple su objetivo de aprender y mejorar, lo cual incluye una fase de pruebas intensiva.

Evidentemente todos y cada uno de los puntos anteriores están pensados de cara a resolver el problema formulado en el apartado anterior; dicho de otra manera, es todo lo que se debe alcanzar para lograr construir un bot que juegue contra el usuario y aprenda y mejore a medida que lo hace, para poder seguir presentando un desafío contra el propio jugador.

1.2.2 Actores implicados

Uno de los principales actores implicados es, evidentemente, el usuario. Dicho de otra manera, el jugador: la persona que jugará y se enfrentará al bot, y al mismo tiempo intentará mejorar para poder seguir ganando. Para ello es importante que pueda aprender los controles básicos del juego mediante algún tipo de tutorial; a partir de ahí, dependerá de su propio nivel de habilidad averiguar hasta dónde puede llegar, y empujar sus propios límites para mejorar todo lo posible, al igual que hará el bot por su lado.

Para este tipo de actor implicado, el rango de personas que engloba es bastante amplio. Desde aquellos jugadores que solo buscan entretenimiento o diversión, hasta los más dedicados que busquen siempre un desafío más allá de su propia habilidad, pasando por personas que simplemente disfrutan probando cosas nuevas.

El otro actor implicado, tal vez no tan evidente, pero muy importante en este proyecto, es el propio bot. Es, al igual que el jugador, una parte fundamental del

juego. Sin el bot, el desarrollo de este proyecto pierde su sentido, ya que el enfoque principal es la implementación del mismo, con el objetivo de que siempre haya un desafío mayor para los jugadores. Por ello, es conveniente considerar el bot uno de los actores implicados principales del proyecto.

A partir de aquí, podemos observar algunos actores secundarios, probablemente prescindibles para el proyecto, pero interesantes para tener en cuenta en este apartado. Uno de ellos, serían los espectadores. De la misma manera que el juego es entretenimiento para el jugador, también puede considerarse entretenimiento para aquellas personas que quieran observar mientras se juega, y por lo tanto se pueden considerar actores. Evidentemente, a diferencia del jugador y el bot, no son imprescindibles.

Por otro lado, podemos encontrar una categoría de actores derivada de los jugadores. Serían los *entrenadores*. Éstos serían jugadores expertos que se dedican a entrenar al bot a niveles altos para que luego éste pueda presentar un gran desafío para jugadores nuevos o inexpertos. Estos mismos entrenadores podrían ayudar también a dichos jugadores a aprender, enseñando tácticas o estrategias más avanzadas.

1.2.3 Influencias

Es útil a la hora de desarrollar un proyecto de este calibre, conocer otros proyectos pasados que utilicen técnicas de inteligencia artificial similares a las que se tendrán que implementar.

Algunos ejemplos interesantes[6] serían *Black & White* de Lionhead Studios, lanzado en 2001, juego en el que tienes algo similar a una mascota que realiza acciones sobre un entorno, y el jugador debe recompensar o penalizar dichas acciones para que en el futuro su mascota actúe mejor. Sería un buen ejemplo de aprendizaje por refuerzo aplicado a videojuegos, con la característica particular de que tanto las recompensas como las penalizaciones son determinadas por el propio jugador en tiempo real.

[6] Ejemplos obtenidos de [1]

Otro ejemplo sería *Creatures* de Millenium Interactive, lanzado en 1996, donde el jugador debe ayudar a animales a sobrevivir, enseñándoles a alimentarse, comunicarse, y defenderse. Fue la primera aplicación popular de aprendizaje automático y redes neuronales en una simulación interactiva.

Un ejemplo más reciente, y a la vez más cercano a los objetivos de este proyecto, es el *Amiibo* de Nintendo, para el videojuego de Nintendo 3DS *Super Smash Bros*. Se trata de un juego de lucha cuyo objetivo es tirar a los oponentes del escenario. El *Amiibo* es una figura pequeña que se conecta a la consola por NFC, y te permite jugar contra un bot que va aprendiendo a cada combate, y almacena los datos y la información aprendida en un chip dentro del propio *Amiibo*. Esto le permite jugar cada vez mejor, e ir subiendo de nivel a medida que lo hace.

1.2.4 Alcance del proyecto

Es importante analizar el procedimiento a seguir para desarrollar este proyecto. Es evidente que hay dos partes principales y diferenciables: El desarrollo del juego y la implementación del bot. Pero no es lo único que habrá que trabajar.

Primero de todo, hay que hacer un análisis teórico de las necesidades para implementar el bot. Esto es necesario ya que a la hora de desarrollar el videojuego, hay que tener en cuenta que habrá que trabajar sobre su implementación a la hora de programar la IA, y si no se ha tenido en cuenta inicialmente, la segunda tarea puede ser mucho más compleja.

Una vez esta fase está completada, es el momento de empezar a implementar el propio videojuego. Primero se diseñarán todos los aspectos de jugabilidad y atributos del mismo, tales como número de personajes, escenarios, tipos de movimientos, condiciones de victoria, etc. Una vez esté todo establecido, se implementará una versión prototipo del juego con un personaje y un escenario, para probar que la estructura base del juego funciona correctamente.

Una vez el prototipo esté completado, se procederá a expandir el contenido del juego, añadiendo personajes y escenarios, y añadiendo los menús adecuados. Para esta parte habrá que buscar fuentes de arte libres para poder ser usados en el

juego, y opcionalmente desarrollar algún aspecto artístico original para el juego, si se da la necesidad y hay tiempo para ello.

En este punto, se deberá probar que todos los aspectos importantes del juego funcionan correctamente, y una vez comprobado el buen funcionamiento, se procederá a la implementación de la IA.

Primero se establecerán unas bases de habilidad para el bot, como si se tratara de un jugador nuevo: Los movimientos básicos y los conocimientos sobre el funcionamiento del juego. Será a partir de estos estados y acciones básicas sobre los que se trabajará la IA a desarrollar, y en ellos se basará el aprendizaje por refuerzo.

Finalmente se procederá a diversas fases de prueba del bot, modificando los parámetros como sea conveniente, y arreglando los errores que puedan surgir. Una vez llegados a este punto, se procederá a entrenar varios bots diferentes, y si hay tiempo, se probará a enfrentar bots entre ellos para que aprendan entre ellos, a modo de experimento adicional.

1.2.4.1 Posibles obstáculos

Los obstáculos que puedan aparecer en el desarrollo de este proyecto son similares a los que nos podríamos encontrar en el desarrollo de cualquier proyecto que requiera desarrollar un sistema informático. A continuación se listan algunos de los obstáculos más notables:

1. Errores en el código: el error más frecuente a la hora de implementar un programa. Es realmente sencillo, y más cuando se trata de un proyecto grande, cometer errores en algunos puntos del código que hacen que el programa funcione de forma inesperada. Todo error requerirá ser encontrado y reparado, lo cual liga directamente con el siguiente obstáculo.
2. Problemas de tiempo: El desarrollo del proyecto tiene un límite de tiempo (básicamente, la entrega final y la defensa del trabajo). Esto implica que se debe desarrollar dentro del límite, y aunque se haga una planificación aproximada del tiempo que tomará cada parte del desarrollo, pueden surgir

contratiempos (como los errores comentados en el punto anterior) que aprieten el calendario de entrega.

3. Funcionamiento inesperado: En el proyecto se va a desarrollar una inteligencia artificial que utilice técnicas de aprendizaje por refuerzo. Pero eso no significa que los algoritmos vayan a funcionar tal cual se describen en la documentación de los mismos, ya que hay que adaptarlos al entorno del juego. Si se comete algún error conceptual a la hora de determinar estos algoritmos, el funcionamiento del bot no será el esperado, y habrá que dedicar tiempo extra a arreglar los posibles errores conceptuales que se hayan cometido.

1.2.5 Alcance de la memoria

En cuanto a la memoria de este proyecto, se ha intentado cumplir los siguientes puntos establecidos:

- Dar una introducción relevante para la escala del proyecto, y que sirva de nexo entre el lector y los datos más técnicos del mismo.
- Ofrecer una descripción del videojuego y de todos sus aspectos, tanto visuales como de lógica, al mismo tiempo que describe las técnicas y herramientas utilizadas.
- Detallar toda la implementación del código, tanto del juego como de la IA.
- Explicar y analizar los resultados obtenidos tras todas las pruebas.
- Finalizar la lectura de la misma con un breve apartado de conclusiones.

1.3 Estado del arte

1.3.1 Videojuegos e inteligencia artificial

Como se ha comentado anteriormente, los videojuegos y la inteligencia artificial van generalmente cogidos de la mano. Es cierto que hay excepciones, como por ejemplo juegos puramente controlados por los jugadores (algunos juegos

competitivos no tienen modo *single player*[7]), pero por norma general todos los juegos incluyen técnicas de IA en alguno de sus aspectos.

Por ejemplo, imaginemos un videojuego en el cual hay enemigos que deben perseguir al jugador por un escenario con obstáculos y paredes. Los enemigos deberán tener implementada una IA que les permita realizar la persecución esquivando los obstáculos y las paredes. Y si queremos ir un poco más allá, además querrán realizar el camino óptimo durante dicha persecución. Para ello habría que implementar una versión del algoritmo A*[8], por ejemplo, aplicada sobre un grafo de nodos, o incluso sobre un mapa de polígonos[9] para más precisión.

A lo largo de la historia, las IA de los videojuegos han ido evolucionando a medida que la demanda lo requería. Dicho de otra manera, los videojuegos progresaron durante gran parte de su historia en la dirección de jugarse en solitario, y no en compañía de amigos. Inicialmente la mayoría de juegos estaban enfocados a ser jugados con más personas; si pensamos en las máquinas *arcade* que solía haber en los inicios de los salones recreativos, la mayoría de juegos permitían el control de varios jugadores.

Pero a medida que empezaron a popularizarse las videoconsolas, los videojuegos empezaron a convertirse en algo más “individual”, ya que era algo que cada uno podía jugar sólo en su casa. Esto requería que las IA de dichos juegos fueran algo más complejas de lo que venían siendo hasta entonces, ya que tenían que ser capaces de ofrecer al jugador el entretenimiento y el desafío que buscaba (al fin y al cabo es el objetivo final de los videojuegos, el de entretener y divertir).

Es interesante comentar que a pesar de la evolución de las IA de los videojuegos a lo largo de estos años, hoy en día, algunos desarrolladores han vuelto a los orígenes de los juegos competitivos en los que los bots no tienen importancia, en

[7] Es un modo de juego que, como su nombre bien indica, es para un solo jugador, lo cual implica que probablemente haya algún aspecto del modo de juego controlado por una IA para interactuar con el jugador.

[8] Algoritmo de búsqueda de *shortest paths* mediante heurísticas. Referencia a [6]

[9] En videojuegos, se puede establecer un mapa de polígonos que son adyacentes los unos con los otros, y que en su totalidad definen el espacio por el que los personajes pueden moverse.

forma de *juegos en línea*, que permiten jugar desde casa con amigos, contra otros jugadores, sin necesidad de enfrentarse a la máquina.

Con el auge de los *e-sports*[10], cada vez se le da menos importancia a la IA en los juegos de tipo competitivo, ya que hemos regresado a la era en la que los jugadores pueden jugar y competir entre ellos, sin necesidad de bots que jueguen contra ellos.

1.3.2 Desarrollo de videojuegos

En los inicios, los videojuegos eran algo bastante poco común y desarrollado por algunas pocas empresas especializadas. Poco a poco se fueron popularizando y fueron apareciendo nuevas empresas que se dedicaban a su desarrollo, y hace años que empezaron a surgir incluso desarrolladores independientes, que creaban juegos sin el soporte empresarial que venía dado hasta el momento para su desarrollo.

A día de hoy, cualquiera que esté interesado en el tema puede intentar aprender a desarrollar videojuegos con tiempo y dedicación, gracias a herramientas y motores como Unity o Unreal Engine[11], que son accesibles para todo el mundo, y simplifican bastante el proceso de creación de videojuegos (ya que te traen una base funcional predefinida, sobre la cual implementas tu propio juego).

Así pues, actualmente hay una gran cantidad de *Game Developers*[12], tanto profesionales como aficionados, y es por ello que una gran cantidad de videojuegos de todo tipo salen al mercado a diario, algunos muy simples, otros más complejos. Eso no elimina el hecho de que desarrollar un videojuego de manera correcta, obteniendo un resultado aceptable, sigue requiriendo conocimientos de todos los aspectos que lo forman: desde *game design*[13], hasta el apartado artístico, pasando por programación, pruebas, y balanceo[14].

[10] Deportes electrónicos, es el nombre que reciben los juegos competitivos en línea hoy en día, ya que hay todo un mundo profesional alrededor de ellos.
[11] Referencias: [7,8]
[12] Desarrolladores de juegos, independientemente del apartado que trabajen (arte, programación…).
[13] El diseño del juego trae bastante trabajo debido a que es lo que denotará todas las características del juego final.
[14] Balancear un juego se refiere a modificar los parámetros de manera adecuada, y es necesario para encontrar un equilibrio correcto entre dificultad y diversión.

1.3.3 Aprendizaje por refuerzo

Hace bastantes años que se estudian técnicas de aprendizaje por refuerzo en el mundo de la inteligencia artificial. A lo largo de los años se han ido desarrollando nuevos algoritmos y técnicas aplicables a este tipo de aprendizaje automático, y hoy en día se puede encontrar no solo en juegos, sino en todo tipo de sistemas inteligentes.

La idea básica tras las técnicas de aprendizaje por refuerzo se basa en, dado un estado y un listado de acciones que cambian el estado, y que al mismo tiempo reciben una recompensa o penalización del entorno, buscar la política de acciones que optimice la suma de las recompensas.

Hay múltiples variaciones de esta idea, y distintos algoritmos que la ponen en práctica, como por ejemplo *Q-Learning*, que se basa en calcular para cada estado un valor determinado por las recompensas obtenidas en los estados a los que se pueden llegar desde él. Tiene sus ventajas y sus limitaciones, pero es aplicable a distintos tipos de juegos[15].

En este proyecto, se hará un estudio y análisis de las distintas técnicas y algoritmos conocidos, y se desarrollará un sistema inteligente que sea adecuado para el estilo de juego, y que permita al bot aprender de manera sólida partida tras partida.

1.3.4 Leyes y regulaciones

1.3.4.1 PEGI

Para publicar videojuegos en toda el área Europea, es necesario cumplimentar con las regulaciones establecidas por PEGI (Pan European Game Information) [9]. Estas regulaciones se basan en dos niveles de información:

- Las etiquetas de edad de PEGI: Consta de 5 categorías, determinadas por la edad mínima para jugar al videojuego. Estas categorías son “mayores de 3, 7, 12, 16 y 18 años”. Todo videojuego publicado en Europa requiere esta etiqueta.

[15] Referencias: [2] capítulo 2.6

- Los descriptores de contenido: Son 7 etiquetas adicionales y opcionales que describen contenido de una índole específica, y que está contenido en el videojuego. Estas etiquetas son: “Violencia, lenguaje soez, miedo, juego (en cuanto a casino se refiere), sexo, drogas y discriminación”.

Dadas estas categorías, el videojuego desarrollado en este proyecto se podría catalogar como PEGI 7 con la etiqueta adicional de “violencia”.

1.3.4.2 ESRB

Para publicar videojuegos en toda el área de América (especialmente Norteamérica), es necesario cumplimentar con las regulaciones establecidas por ESRB (Entertainment Software Rating Board) [10]. Estas regulaciones también se basan en los dos niveles de información de PEGI, con algunas diferencias:

- Las etiquetas de edad en ESRB son las siguientes: “Everyone” para todos los públicos, “Everyone +10” para mayores de 10 años, “Teen” para mayores de 13 años, “Mature” para mayores de 17 años, “Adults only” para mayores de 18, y “Rating Pending” en caso de que todavía no se haya establecido categoría (material promocional, por ejemplo).
- Los descriptores de contenido de ESRB son mucho más extensos que los de PEGI, e incluyen estos como un subconjunto del mismo. Para una lista completa, consultar [10].

En este caso, el videojuego se incluiría en la categoría “ESRB Everyone”, con el descriptor de “violencia”.

1.3.4.3 Otros

En caso de querer publicar en determinadas regiones del planeta (Brasil, Asia, China, Japón, etc.) es necesario consultar directamente con las empresas de *publishing* locales, ya que serán éstas las que se encarguen del proceso de regulación y adaptación del contenido en sus regiones.

1.4 Estructura del documento

Este documento está dividido en diversas partes, que en conjunto toman forma y explican de forma detallada el proceso seguido para el desarrollo del trabajo. Estas partes son las que se explican a continuación:

- Descripción del videojuego: En este apartado se explicará todo el diseño del videojuego, así como el funcionamiento del mismo:
  - Menús, jugabilidad, personajes, escenarios, etc.
  - Mecánicas del juego
  - Diseño de la IA para el bot
- Implementación del videojuego: En esta sección se explicará todo lo relacionado con el código y datos relevantes para la implementación del juego:
  - Programación de menús
  - Implementación de personajes
  - Creación de escenarios
  - Implementación del bot (algoritmos, políticas de decisiones).
- Análisis de resultados: Dadas las pruebas que se realizarán en base a lo establecido en el diseño de la IA, se obtendrán y analizarán los resultados que se generen a lo largo del aprendizaje del bot bajo distintas estrategias.
- Conclusiones: Apartado final para cerrar el documento y comentar posibles alternativas y extensiones para el proyecto.

2. Descripción del videojuego

En las próximas secciones del documento haremos un recorrido por todas las partes de las que consta el videojuego, que incluirán tanto el diseño del mismo como el de la IA que controlará los bots.

2.1 Interfaz

En la pantalla de juego, se pueden observar distintos elementos:

Figura 2.1: Imagen de la interfaz de juego

1. Barra de vida del personaje izquierdo (el controlado por el jugador).
2. Barra de *stamina* del personaje izquierdo.
3. Barra de vida del personaje derecho (controlado por un bot o un segundo jugador.
4. Barra de *stamina* del personaje derecho.
5. Personaje controlado por el jugador.
6. Personaje controlado por un bot o un segundo jugador.

Las barras de vida son indicadores del estado actual del personaje en el combate. A más daño recibido, menor es el valor de esta barra. Cuando una de las dos barras

de vida llega a 0, se acaba el combate, siendo perdedor el personaje que se haya quedado sin vida.

Las barras de *stamina* son indicadores de las habilidades que pueden utilizar los personajes en cada momento. Suben y bajan en función de las acciones de cada uno.

Cada uno de los elementos marcados en la figura 2.1 van a estar en constante movimiento: tanto los personajes porque son los actores principales del combate, como sus respectivas barras, que van a estar cambiando en función del estado de los personajes. Además, la imagen de fondo del escenario también está animada, con lo cual habrá movimiento constante en la pantalla de juego.

2.2 Jugador

2.2.1 Jugabilidad

Cuando se va a diseñar el jugador para un juego de lucha, hay dos partes principales a tener en cuenta: movimientos y acciones. Mientras que lo primero controla el posicionamiento del personaje en el escenario, el segundo es el que hace que el juego progrese. Vamos a analizarlas más en detalle.

2.2.1.1 Movimiento

El movimiento es todo el conjunto de habilidades que permiten al personaje cambiar de posición. En el caso de este juego, estas habilidades son las siguientes:

- Movimiento horizontal estándar: El movimiento básico del personaje, hacia la derecha o izquierda. Corre a velocidad fija en un movimiento contínuo. No gasta ningún recurso del personaje.
- Dash (carga): Es un movimiento especial que permite al personaje avanzar o retroceder rápidamente usando un pequeño impulso, que puede usarse para esquivar ataques o para acercarse rápido al rival. El dash requiere un coste de stamina.

- Salto: Un movimiento que permite despegar del suelo para moverse por el aire temporalmente. Mientras se está en el aire, la gravedad hace que el personaje vaya cayendo. Hay diversas acciones que no se pueden realizar mientras se está saltando (por ejemplo, bloquear), pero en cambio hay otras que funcionan mejor (el dash, entre otros). Saltar no gasta ningún recurso del personaje, pero se pausa la recuperación de stamina mientras el personaje está en el aire.
- Agacharse: Un movimiento simple que mantiene al personaje cerca del suelo. Las acciones que se pueden realizar mientras se está en este estado son prácticamente las mismas que estando de pie, pero con menos eficaces en general, a excepción del bloqueo, que es más efectivo.

2.2.1.2 Acciones

Las acciones son habilidades relacionadas con el combate y que harán que la partida pueda progresar, ya que con el movimiento no se puede avanzar hacia el final (la vida de los personajes no bajan).

Primero de todo, las acciones más importantes: las acciones de ataque. Se dividen en tres categorías:

- Puñetazos: Son ataques simples que no hacen mucho daño, pero son rápidos y difíciles de contrarrestar, con lo cual son un método perfecto para bajar la vida del rival de forma consistente, y de romperle el bloqueo.
- Patadas: Son ataques más complejos que los puñetazos, que a la vez hacen más daño, pero también son más fáciles de contrarrestar usando bloqueo y contraataque. Son un método perfecto para rebajar la vida del rival rápido cuando le has roto el bloqueo.
- Ataques de rango: Son unos proyectiles que se pueden disparar desde la distancia. No hacen mucho daño, y gastan stamina al ser disparados, pero son una buena manera de desgastar el bloqueo y la concentración del rival.

Los puñetazos y las patadas están divididas en conjuntos de 3 golpes, que al efectuarlos de manera continuada, forman un *combo* (combinación de golpes). De

manera que pegar 3 puñetazos seguidos o 3 patadas seguidas formarán un combo. Pero de la misma manera, también se pueden combinar puñetazos y patadas en un mismo combo, por ejemplo, si se efectúa la combinación “patada-puñetazo-patada”, se habrá realizado un combo con una patada seguida de un puñetazo y acabado con otra patada. Los ataques de rango, en cambio, no forman parte de ningún tipo de combo. En el apartado de mecánicas del juego se explicará en más detalle el funcionamiento de los combos.

A continuación se explicarán las acciones defensivas, que son dos: bloqueo y contraataque.

- El bloqueo es una acción que permite protegerse de los ataques del rival. Se puede ejecutar tanto estando de pie como agachado, pero no mientras se salta o se efectúa un dash.
- El contraataque es una acción especial que puede realizarse al bloquear un ataque rival, y que permite causarle grandes daños al atacante si su stamina es lo suficientemente baja.

Dominar ambas acciones será clave para obtener la victoria en este videojuego, dadas las mecánicas de juego relacionadas con ellas, que se explicarán más adelante.

2.2.2 Controles

Se han diseñado controles tanto para teclado como para gamepad (mando de Xbox 360). Esto es para que cualquier persona pueda adaptarse a sus preferencias, ya que muchas personas están más acostumbradas a jugar con teclado, y otras en cambio prefieren jugar con un gamepad.

Los controles de teclado están definidos de la forma siguiente:

- A,D: Mover al personaje horizontalmente
- S: Agacharse
- Espacio: Saltar
- J: Puñetazo

- K: Patada
- L: Ataque de rango
- U: Contraataque
- Shift: Bloquear/Dash

Los controles en el mando de Xbox 360 están definidos de la forma siguiente:

- Joystick izquierdo/D-pad: Mover al personaje horizontalmente
- RT: Agacharse
- A: Saltar
- X: Puñetazo
- Y: Patada
- B: Ataque de rango
- LB: Contraataque
- RB: Bloquear/Dash

Para definirlos, se han propuesto diversos diseños posibles, y se ha escogido el que resulta más cómodo y similar a los controles de juegos ya existentes. El objetivo es que sea sencillo de manejar y no genere demasiado cansancio al jugar.

2.2.3 Apartado artístico

Por último, pero no menos importante, está el apartado artístico (visual). Es un apartado imprescindible en todo videojuego, debido a que es lo que los jugadores verán al jugar. Es por ello que hay que darle importancia al arte usado para todo, en especial para los personajes.

En el caso de un videojuego de lucha en 2D, hay que utilizar *sprites* para representar a un personaje. Un *sprite* es un dibujo 2D que describe visualmente al personaje, y que contiene una secuencia de fotogramas que representan las animaciones del personaje. Estas animaciones son, entre otras:

- El personaje estático, es decir, mientras está de pie sin moverse ni realizar ninguna acción.

- Las animaciones de movimiento, ya sean de correr, caminar, saltar, hacer dash, o agacharse.
- Las animaciones de cada acción, ya sea de los golpes (patada, puñetazo), ataque de rango, bloqueo, contraataque, etc.

Todas las acciones y los movimientos que tenga un personaje requieren su animación respectiva, de manera que el jugador tenga *feedback* visual de lo que está ocurriendo en el juego en todo momento.

2.3 Mecánicas del juego

Es necesario describir todo el funcionamiento del juego, lo que se suele llamar *mecánicas del juego*. Esto incluye los objetivos de una partida, y cómo se desarrolla la misma. Empezaremos analizando los estados de los personajes: la vida y la stamina.

El objetivo de un combate es reducir la vida del rival a 0 al mismo tiempo que evitas que se reduzca la tuya a 0. La vida se reduce recibiendo daño por golpes o ataques del rival, y en función del ataque bajará más o menos. Además, muchas de estas acciones requieren stamina, que es el valor que determina qué acciones puede realizar un personaje en un momento dado. A diferencia de la vida, la stamina puede llegar a 0 sin repercusiones directas en el combate, y además se va regenerando a medida que pasa el tiempo, mientras que la vida no se recupera de ninguna manera.

La manipulación de stamina dependerá del estado y de las acciones del personaje. Determinadas acciones harán que la stamina se reduzca, y otras acciones harán que esta no se recupere. Es por ello que hay que definir todas las mecánicas relacionadas con esto. Empezaremos por el movimiento.

El movimiento horizontal estándar del personaje no gasta stamina, y tampoco bloquea la recuperación de stamina, con lo cual se podría decir que es completamente independiente de esta. Los movimientos que sí afectan a la stamina son:

- El dash, que requiere un 25% de stamina para efectuarse.

- El salto, que hace que no se regenere la stamina mientras el personaje está en el aire.

En cambio, ningún movimiento tiene efecto sobre la vida de los personajes, mientras que las acciones de ataque están directamente relacionadas con esta. En caso de que uno de los 2 tipos de ataque directo (puñetazo, patada) no sea bloqueado, reducirá la vida del rival siguiendo la fórmula siguiente:

$$D_{final} = D_{golpe} * \%Stamina + D_{mínimo}$$

Donde D_{final} es el daño final que recibirá el rival, D_{golpe} es el daño asignado a cada golpe en particular, %Stamina es el porcentaje de stamina del personaje que ataca, en el momento que ataca, y $D_{mínimo}$ es el daño mínimo que hacen todos los ataques (dicho de otra manera, el daño que haría el golpe si la stamina del personaje estuviera a 0). Los ataques de rango en cambio, no siguen ninguna fórmula, simplemente hacen daño fijo en caso de no ser bloqueados.

En caso de ser bloqueado, el resultado cambia. Mientras se está bloqueando, la stamina no se recupera, de manera que hay que ir con cuidado al bloquear. Además, a cada golpe bloqueado, se reduce la stamina de ambos personajes, en función del daño del golpe. Cuanto más baja sea la stamina del personaje que bloquea, más probabilidades hay de que el bloqueo se rompa, causando que el personaje se quede al alcance de cualquier ataque enemigo entrante. Es por ello que aprender a bloquear bien es clave para mejorar en el juego.

Por otro lado, el contraataque depende del bloqueo. Solo se puede efectuar un contraataque tras bloquear el ataque de un rival que tiene la stamina baja. En particular:

- Se puede contraatacar un puñetazo si la stamina del atacante se encuentra bajo el 20%.
- Se puede contraatacar una patada si la stamina del atacante se encuentra bajo el 35%.

No se pueden contraatacar los ataques de rango, ni los ataques realizados desde el aire.

A continuación, se introducirán los conceptos de *finisher* y *knockback*.

- *knockback* es cuando un personaje es empujado hacia atrás a la fuerza al recibir ciertos golpes. Mientras estás siendo empujado por el knockback, no hay ningún tipo de control sobre el personaje, hasta que no se recupera del empujón.
- *finisher* es el nombre que se le da a golpes finales de combo, que a su vez causan knockback en el contrincante.

Los ataques que se comportan como finishers en este juego son los siguientes:

- El tercer puñetazo del combo.
- La tercera patada del combo.
- El contraataque.
- El golpe final del combo mientras está el personaje agachado
- Cualquier golpe realizado desde el aire.

Además, los finishers suelen ser más fuertes que los demás ataques (D_{golpe} mayor).

2.4 Modos

El aprendizaje de la IA se efectuará a partir de un plugin externo al propio juego, que se comunicará con este a través de un socket del PC. Es por esta razón que se debe dividir el juego en dos modos: el modo normal, y el modo de entrenamiento.

2.4.1 Modo normal

El modo normal del juego sería el que cualquier usuario utilizará normalmente. Se basará en jugar a combates normales contra otro jugador o contra un bot ya entrenado, con pausas entre combates, y menús para seleccionar las opciones necesarias.

Además, en este modo, los combates seguirán las reglas establecidas y funcionarán a una velocidad estándar, haciendo que sea el modo de juego preferible para pasar el rato.

2.4.2 Modo entrenamiento

El modo entrenamiento es un modo de juego especial que está preparado particularmente para el entrenamiento de agentes. Esto significa que seguirá unas reglas especiales que no aplican al modo normal:

- La velocidad de juego está acelerada respecto al modo normal, para acelerar el aprendizaje de los agentes.
- Permite tanto enfrentar a un jugador contra un bot, como a un bot contra otro bot.
- No tiene ningún tipo de menú, y los combates se ejecutan de forma continua, sin pausas (es decir, en cuanto acaba un combate empieza directamente el siguiente).
- Se ejecuta de manera independiente al modo normal.

Es importante ver que, dadas estas restricciones, el modo de entrenamiento debe utilizarse única y exclusivamente para entrenar los agentes que controlarán a los bots, ya que de utilizarlo como modo de juego normal, el nivel de satisfacción general disminuirá y se perderá parte del entretenimiento deseado.

2.5 Diseño de la IA

Una parte muy importante antes de pasar a explicar la implementación del proyecto, es definir cómo funcionará la IA de los bots. Se ha decidido diseñar 3 estrategias distintas, para observar cuál de ellas es la más adecuada para este juego, tanto en velocidad de aprendizaje como en resultado final.

Cada estrategia se basará en un diseño de parámetros de observación y un conjunto de recompensas y penalizaciones, para establecer la política de toma de decisiones del agente. Los parámetros de observación son aquellos atributos del estado del juego que el agente observará y enviará a la red neuronal para analizar el

entorno, que en cada estrategia serán un poco distintos. Cada estrategia intentará mejorar aspectos de la anterior que puedan hacer que el aprendizaje funcione peor de lo esperado.

2.5.1 Q-Learning

Antes de empezar a diseñar las estrategias que se implementarán, es necesario hacer una pequeña introducción teórica a los algoritmos de aprendizaje por refuerzo que se usarán en este proyecto, de manera que se pueda comprender el funcionamiento interno de la IA que se implementará en la sección 3.4.

Q-Learning es el nombre de uno de los algoritmos básicos de aprendizaje por refuerzo. Como ya se ha explicado anteriormente, el aprendizaje por refuerzo se basa en el método que tiene el agente de aprender a partir de recompensas y penalizaciones recibidas del entorno cada vez que realiza acciones para cambiar de estado. El Q-Learning propone un método basado en esta idea, que no depende ni de la política de decisiones que siga el agente ni del entorno en el que se encuentre (*off-policy* y *model-free*).

En particular, el método propuesto es el de, dado un estado *s* y una acción *a* aplicable desde *s*, calcular un valor Q(s,a) para cada combinación de estados y acciones que pueda surgir en una simulación (o en una partida, en nuestro caso), y entonces priorizar la optimización (maximización) de los valores Q (*Q-value*) encontrados. La ecuación que calcula un Q-value es una variante de la ecuación de Bellman:

$$Q(s,a) = E_{s'} [r + \lambda Q(s',a') \mid s,a]$$

La terminología de la ecuación es la siguiente:

- (s,a): estado y acción para los cuales queremos calcular el Q-value
- (s’, a’): estado y acción que llevaron al estado *s*.
- r: recompensa/penalización por ejecutar la acción *a* desde *s*.
- $E_{s'}$: Expectación, valor que indica lo esperable que es llegar al estado s’

- λ: Factor de descuento, indica el peso que tienen Q-values pasados en el cálculo del actual.

Pero esta ecuación por sí misma no da información sobre cómo iterar en la política de decisiones. Para ello establecemos una nueva fórmula para optimizar el Q-value, basada en un parámetro α denominado el *learning rate* (cómo de rápido nos acercamos al objetivo):

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \lambda \max_a Q(s_{t+1}, a) - Q(s_t,a_t)]$$

Usando esta fórmula, vamos actualizando los Q-values conocidos mediante nuevos Q-values que vayamos obteniendo a medida que progresa el algoritmo de aprendizaje, de manera que poco a poco logremos obtener Q-values óptimos para cada estado y acción posibles.

2.5.2 La variante que usaremos: PPO

Para muchos casos, calcular el Q-value no es suficiente. Por ejemplo imaginemos varias acciones que nos dan la misma recompensa, pero alguna de ellas nos da más ventaja de cara al futuro. Para ello, es necesario definir esta ventaja de la siguiente manera:

$$A(s,a) = Q(s,a) - V(s)$$

Donde V(s) es la media de todos los Q-values del estado *s*. La ventaja no deja de ser un valor que determina cómo de bueno es un Q-value en comparación a la media de los Q-values de su estado.

Existe un algoritmo que utiliza la ventaja para calcular una función objetivo que nos indique los cambios que se han de realizar en la política de decisiones del agente: *Trust Region Policy Optimization* (TRPO). El problema que tiene este algoritmo es que dada la formulación utilizada para calcular dicha función objetivo, a pesar de que los resultados que brinda son excelentes, su computación e implementación es extremadamente compleja. Es por ello que se desarrolló una variante de TRPO que

cambia la formulación de dicha función objetivo para simplificar su implementación: *Proximal Policy Optimization* (PPO).

PPO define $r_t(\theta)$ como el ratio entre la política antigua y la nueva, de manera que permita limitar la función objetivo, haciendo que se penalicen grandes cambios en la política de decisiones. De esta manera se simplifica el proceso al establecer un *lower bound* y un *upper bound* al valor de la función objetivo, ya que se podrán computar de forma local los nuevos valores al no haber grandes variaciones entre una iteración y otra del cambio de políticas.

Además, en función de la ventaja de un estado y una acción, el valor de $r_t(\theta)$ puede aumentar o no, pero siempre nos aseguraremos que por mucho que incremente, nunca sobrepasará el límite que hemos establecido. Debido a esto, es interesante comentar que se ha comprobado empíricamente que PPO brinda mejores resultados que TRPO, además de que es mucho más eficiente. [13]

2.5.3 Estrategia 1

Una vez hecha la introducción a los conceptos teóricos, podemos pasar a diseñar las estrategias en sí, empezando por la estrategia inicial. Vamos a intentar observar todo lo que se pueda del entorno para maximizar la información que recibe la red neuronal, y vamos a establecer recompensas y penalizaciones muy detalladas.

Parámetros de observación:

- Distancia al rival
- Velocidad del personaje propio y rival
- Direcciones en las que están mirando los personajes
- Hash de las animaciones que están realizando ambos personajes
- Estado de los personajes (vida y stamina)
- Existencia de algún proyectil dirigido al personaje

Estos parámetros intentan observar todo lo que podría llevar a un jugador real a establecer una estrategia, con lo cual hay mucha información que procesar.

Las recompensas/penalizaciones son las siguientes:

- Recompensa media por reducir la vida del rival
- Recompensa media-pequeña por reducir la stamina del rival
- Penalización media por reducir la vida propia
- Penalización media-pequeña por reducir la stamina propia
- Recompensa grande por ganar la partida
- Penalización grande por perder la partida
- Recompensa pequeña por acercarte al rival
- Penalización pequeña por no hacer nada

Las recompensas y penalizaciones también tienen en cuenta muchos casos que podrían ser considerados relevantes para la modificación del valor de un estado y sus acciones posibles.

2.5.4 Estrategia 2

La primera estrategia tiene en cuenta demasiados parámetros de observación, y algunos de ellos pueden hacer que la función de pesos se vaya por las nubes. A continuación vamos a proponer una estrategia mucho más simple que trate el mínimo de parámetros posible:

- Distancia al rival
- Velocidad de los personajes
- Direcciones en las que están mirando
- Existencia de algún proyectil dirigido al personaje

En cuanto a las recompensas/penalizaciones, no las modificamos tanto como los parámetros de observación, pero sí que se eliminan algunas que complican demasiado el cálculo de pesos:

- Recompensa media por bajar la vida del rival (penalización por reducir la propia)
- Recompensa/penalización pequeña por cambios en la stamina
- Recompensa/penalización grande por ganar/perder la partida

2.5.5 Estrategia 3

Mientras que la primera estrategia era demasiado compleja, la segunda es demasiado simple. Vamos a intentar encontrar un punto medio, con los siguientes parámetros de observación:

- Distancia al rival
- Velocidad de los personajes
- Direcciones en las que están mirando
- Hash normalizado para las animaciones
- Stamina de los personajes
- Existencia de algún proyectil dirigido al personaje

La idea para estos cambios es que, mientras que la vida no es un parámetro tan útil (ya que nos interesa que el agente juegue siempre lo mejor posible independientemente de la vida restante), la stamina es un parámetro que cambia bastante las opciones disponibles para cada jugador (ya que hay acciones que no se pueden realizar sin stamina). Además, es interesante poder controlar las animaciones de los personajes para prevenir los movimientos del rival en todo momento, pero se ha decidido normalizar el valor del hash, ya que con valores tan altos hace que el aprendizaje sea demasiado lento (tal y como se verá en el apartado de resultados).

Las recompensas y penalizaciones son las siguientes:

- Recompensa/penalización media por reducción de vida de ambos personajes
- Recompensa/penalización grande por ganar/perder la partida
- Recompensa/penalización media/pequeña por cambios en stamina propia
- Pequeña penalización por dejar pasar el tiempo sin hacer nada

Se ha decidido eliminar la stamina ya que es un elemento demasiado cambiante como para que se le asigne recompensas o penalizaciones

3. Implementación

El juego se ha implementado usando el motor de juegos *Unity*, que proporciona distintas herramientas para el desarrollo de videojuegos. En las próximas secciones se explicará cómo se han desarrollado las distintas partes del juego dentro del motor, tanto en apartado de scripting como del propio editor.

Unity funciona mediante objetos y componentes. Los objetos son el contenedor de cada entidad que pueda formar parte del juego, mientras que los componentes son atributos que se asignan a cada objeto, dándoles así forma y función (por ejemplo, componentes para visualizar el objeto, para calcular colisiones, o para ejecutar scripts). La idea del funcionamiento de este sistema es similar a un *Data Driven ECS* (Entity-Component-System), pero fundamentalmente diferente en el hecho que se basa en programación orientada a objetos. No es necesario entrar en detalle con respecto a estos conceptos, ya que no forman parte del contexto del proyecto.

3.1 Escena

El primer paso a la hora de desarrollar un juego en Unity es crear una escena básica en la que probar todos los componentes. Una escena es similar en concepto a un "nivel" en un juego: contiene algunos objetos que interactúan entre ellos, y engloban un "espacio" autocontenido, e independiente de los demás.

En este caso, nuestra escena de juego contendrá diversos objetos:

- El player (el personaje que manejará el jugador)
- El personaje rival
- El escenario
- Los "managers" (objetos que administran el funcionamiento de la partida)
- Otros objetos (colisionadores, por ejemplo)
- Objetos necesarios para el funcionamiento de la IA

En las siguientes secciones se va a describir la implementación de estos objetos, y cómo afectan a la escena o al juego.

3.2 Objeto ‘Player’

El objeto principal y más importante de todos, ya que es el que manejará el personaje del juego (tanto el controlado por el jugador como el controlado por el bot). Es importante dedicarle tiempo a su desarrollo ya que es lo que definirá lo bien que funcione el juego en global.

Para empezar, se crea un objeto vacío en la escena, y se le da el nombre de “Player”. Para poder identificarlo fácilmente, se crea un *tag* (etiqueta) de nombre “Player” y se le asigna, de manera que el rival o cualquier otro objeto pueda comprobar que es el personaje del jugador independientemente del nombre que reciba el propio objeto.

A partir de aquí, el objetivo es ir rellenando el objeto “Player” con los distintos componentes que le darán forma. Las secciones siguientes están dedicadas a explicar todos estos componentes.

3.2.1 Sprite

Lo primero que se debe implementar es el aspecto visual, aunque sea tan solo un *placeholder* (sustituto temporal de la versión final). Esto se debe a que para programar toda la lógica, tendremos que ver cómo los controles afectan al personaje, y para ello debemos ser capaces de verlo.

Comúnmente en los videojuegos 2D se utilizan lo que se denominan *sprites*. Los sprites son texturas 2D dibujadas que contienen el arte de cualquier objeto 2D que vaya a utilizar el juego, por ejemplo, personajes, escenarios, objetos, etc. Además, un sprite puede contener diversos fotogramas para un mismo objeto, con la finalidad de definir animaciones (de las cuales se hablará en la siguiente sección).

Para este juego se ha decidido buscar sprites públicos de algún juego ya existente. En particular, se ha descargado una *spritesheet* (imagen 2D con todos los fotogramas del sprite incluídos) del juego *Touhou Hisoutensoku* [11], mediante una web dedicada a descarga de spritesheets de juegos [12].

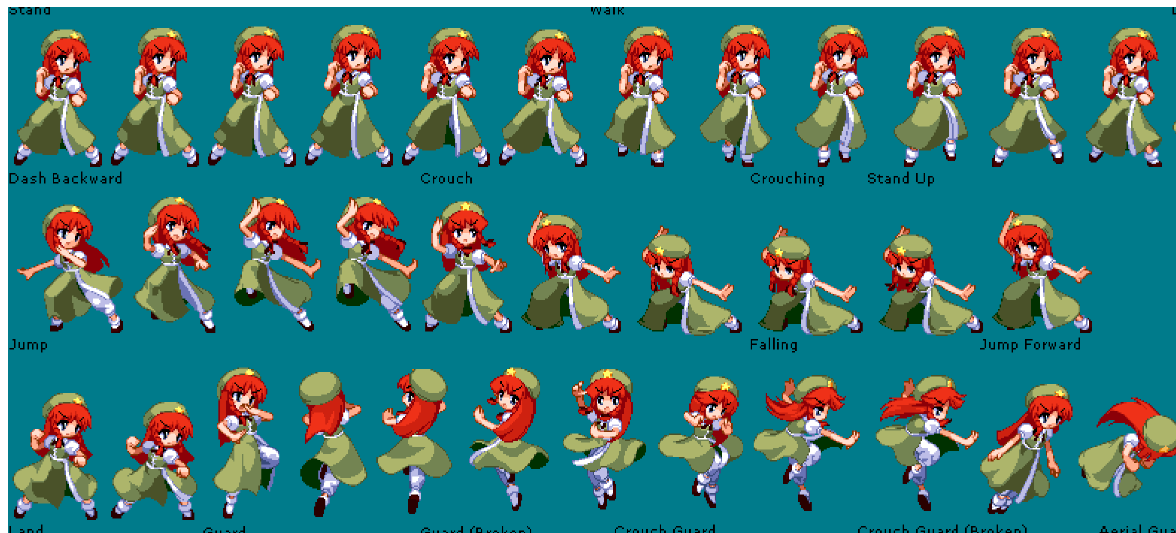


Figura 3.1: fragmento de la spritesheet de personaje usada en el juego

Es importante escoger un spritesheet que sea adecuado para el tipo de juego que se va a desarrollar, y por eso es necesario realizar el diseño del juego antes de empezar a implementarlo. En este caso, el spritesheet de este personaje es perfecto para lo que se debe implementar, de manera que se le añade una componente *Sprite* al objeto del personaje, y se le asigna el spritesheet que se ha comentado anteriormente.

3.2.2 Animaciones

El personaje va a moverse y a realizar acciones a lo largo de la partida. Para ello es necesario que el jugador pueda observar en todo momento lo que le ocurre a su personaje. Esto se puede lograr haciendo uso de *animaciones*.

Las animaciones en 2D son secuencias de fotogramas creados a partir de una o varias imágenes que contienen dichas secuencias. Estas imágenes son los anteriormente mencionados *spritesheets*. En la figura 3.1 se puede observar un fragmento del spritesheet utilizado para este juego.

Por lo tanto, es necesario que una spritesheet contenga los fotogramas de todas las animaciones que nuestro personaje va a necesitar. Cuantas más animaciones se puedan reproducir, más información le daremos al jugador, con lo cual más cómodo de jugar será el juego.

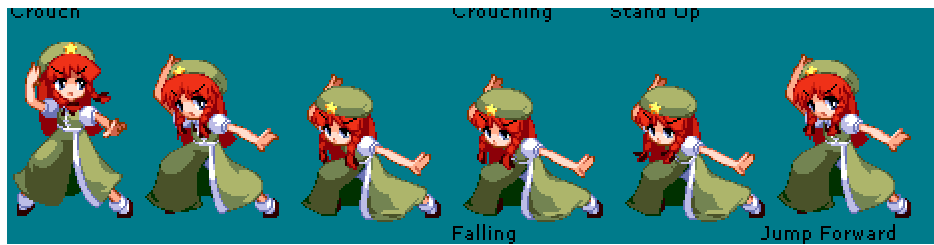
En este caso, nuestro personaje necesitará todas las animaciones que sean relevantes para el diseño de las mecánicas del juego establecidas en la sección 2.3. A continuación se listan las animaciones implementadas, junto con sus fotogramas correspondientes del spritesheet.

- *Idle*: la animación del personaje estando de pie sin hacer nada

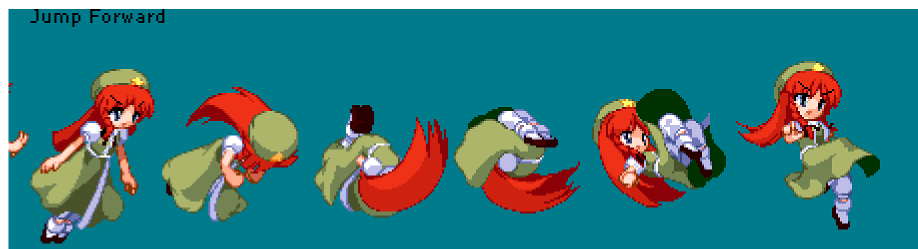
- *Walk*: Caminar hacia adelante

- *Dash*: hacer carga tanto hacia delante como hacia atrás

- *Crouch*: Agacharse

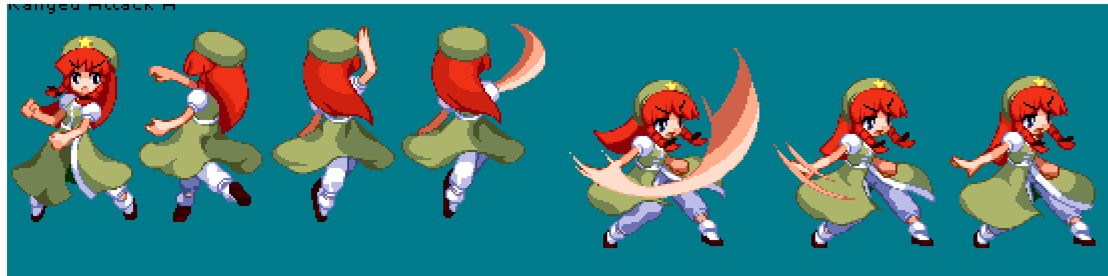


- *Jump*: Saltar, con variante de caída y salto hacia adelante

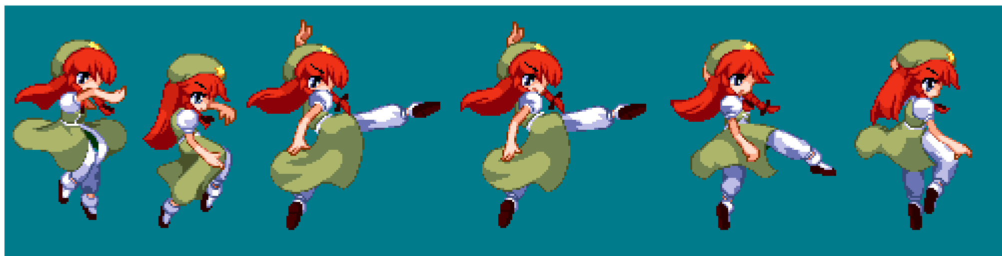


- *Guard*: bloqueo, y todas sus variantes

- *Punch*: Combos de puñetazo



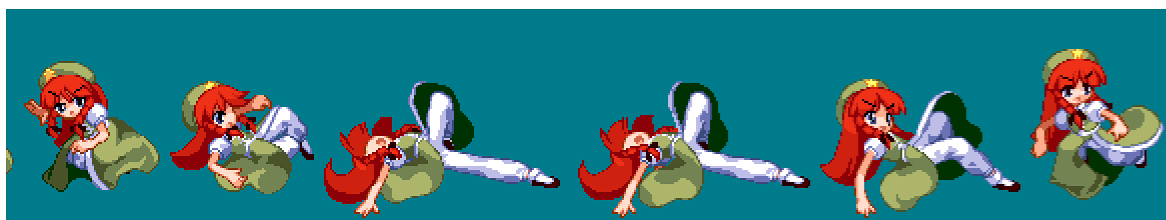
- *Kick*: Combos de patada



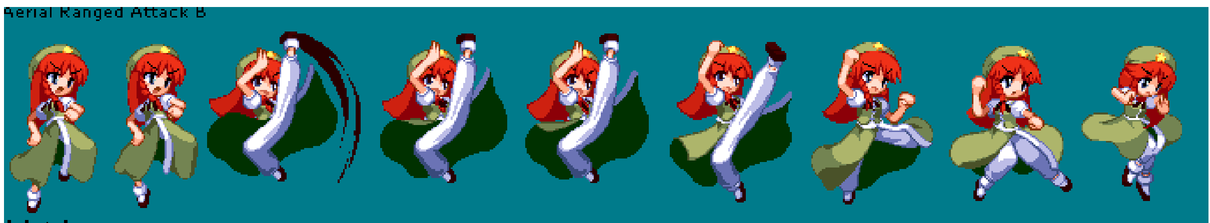
- *Ranged attack*: Ataque de rango

- *Counter*: Contraataque

- *Crouch attacks*: Todos los combos mientras el personaje está agachado



- *Aerial attacks*: Los ataques desde el aire



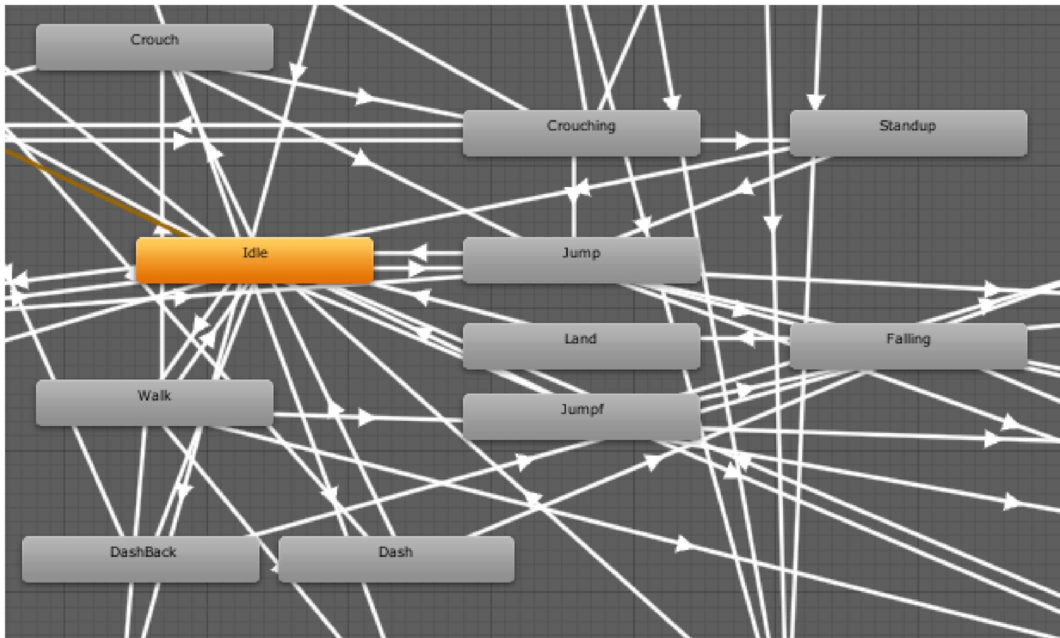
- *Damaged*: Recibiendo daño

- *Knockback*: Al ser lanzado hacia atrás por algún finisher.

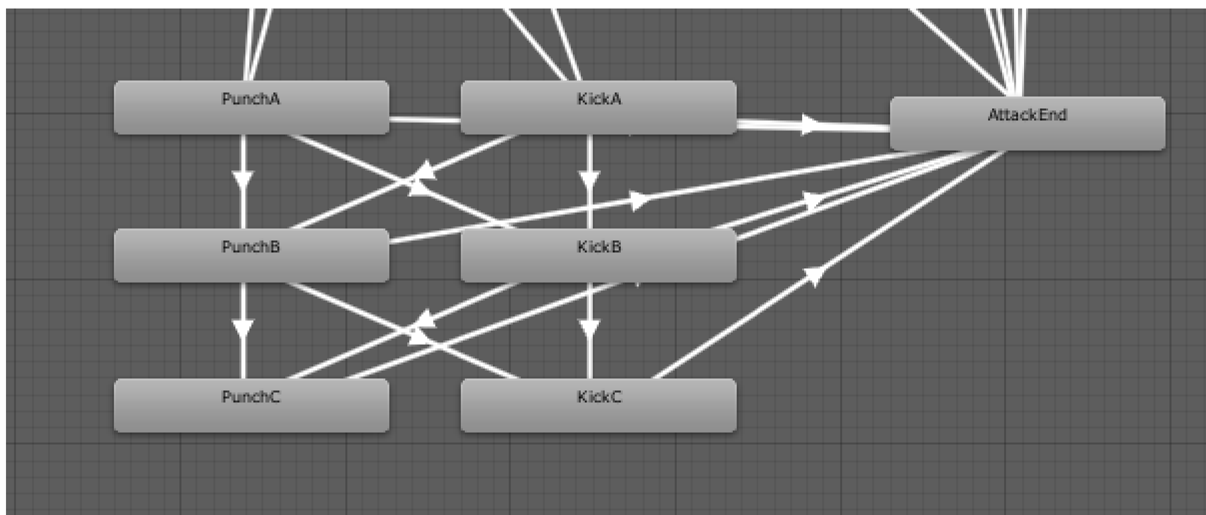
Una vez que las animaciones están definidas (cada una con su secuencia de fotogramas), hay que implementarlas en el juego. Para ello, se ha creado un nuevo objeto de tipo *Animator*, que se utiliza para construir una máquina de estados que definirá el comportamiento de las animaciones del personaje.

Esta máquina de estados contiene diversas partes diferenciadas:

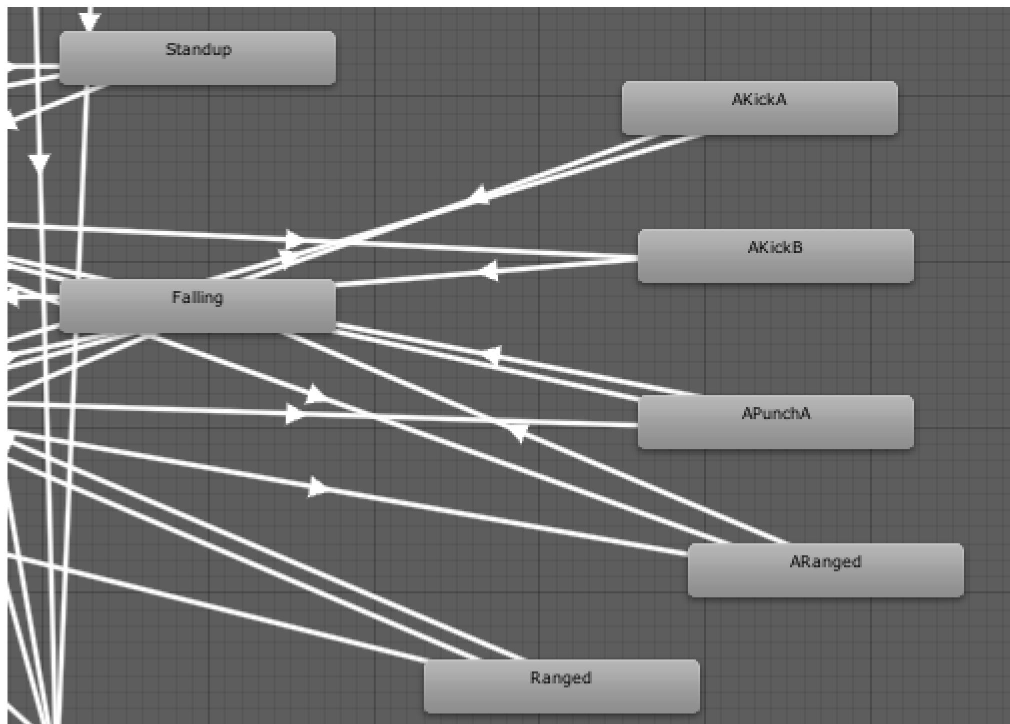
- Estados básicos: Movimiento, Dash, saltar, agacharse. Todas aquellas animaciones que se pueden englobar en el contexto de “animaciones base”.



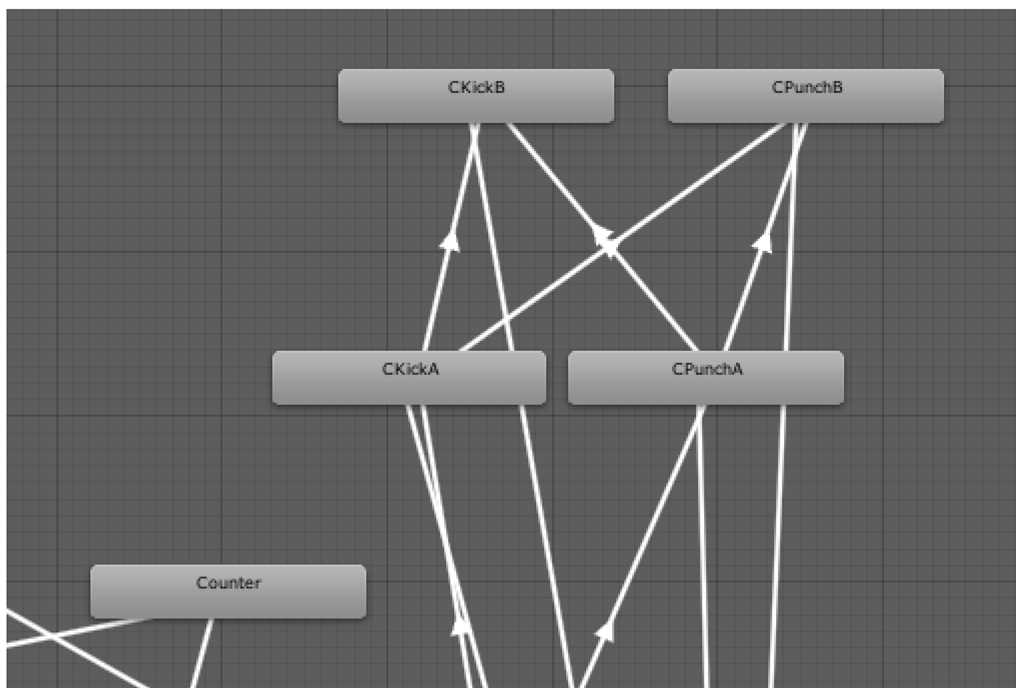
- Estados de combo básico: Las animaciones relevantes para los combos básicos (puñetazos, patadas). Se puede llegar desde la mayoría de estados base, y vuelven a los estados base al acabar.



- Ataques aéreos: Son todos los golpes que se pueden realizar desde el aire. Salen de los estados básicos aéreos (saltos, caídas) y vuelve a estos una vez se acaban.

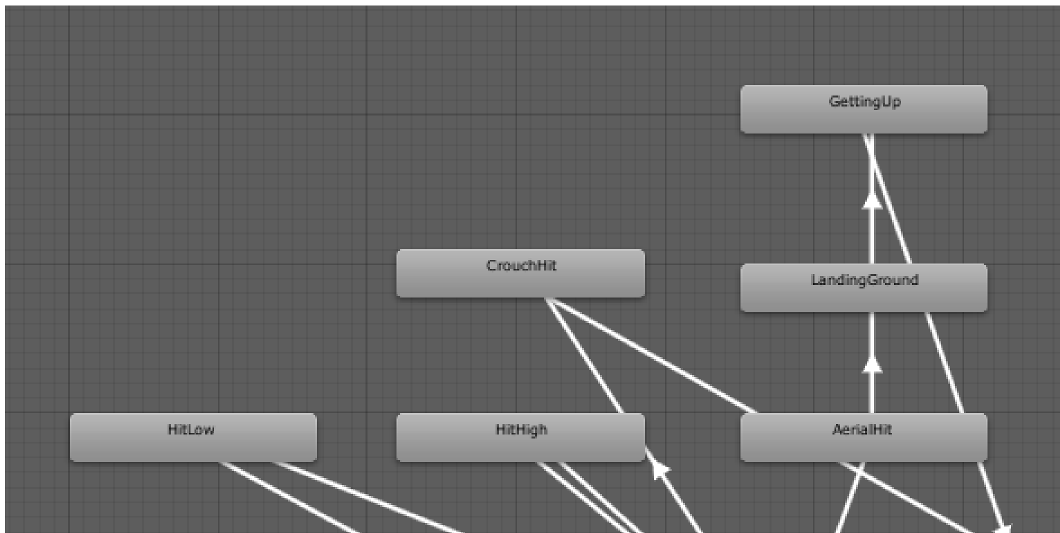


- Ataques agachado: Similar a los combos básicos, también tenemos combos mientras el personaje está agachado, aunque son más simples. Se llega desde los estados de agacharse, y se vuelve a estos al acabar.

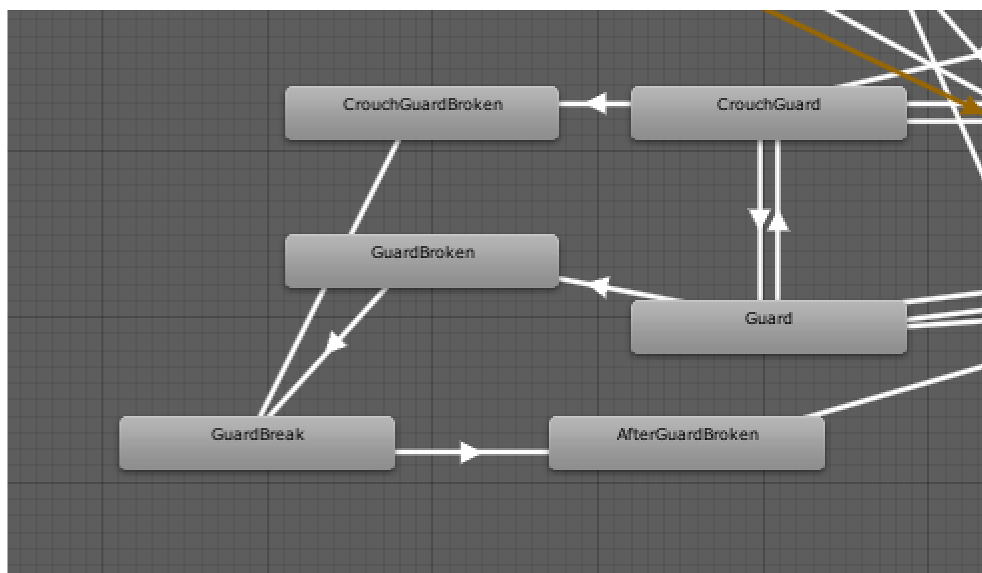


- Recibiendo daño: Hace falta todo un conjunto de estados dedicados a las animaciones de recepción de daño. Son especialmente importantes ya que

nos servirán de bloqueante a la hora de implementar la lógica del personaje, de manera que, por ejemplo, el personaje no pueda moverse mientras está recibiendo daño.



- Bloqueo: La última parte de la máquina de estados está dedicada a las animaciones de bloqueo, y todo lo que deriva de ellas (bloqueo, rotura de bloqueo). A estos estados se llega desde casi cualquier estado base, y se vuelve a Idle después de la rotura de bloqueo (si ocurre).



No es necesario entrar en demasiado detalle acerca de la implementación exacta de la máquina de estados ya que es compleja y no aporta información relevante al

documento; además, en caso de querer examinarla más en profundidad, se puede consultar directamente la máquina de estados en el código fuente del proyecto.

3.2.3 Físicas

Otro aspecto importante a tener en cuenta a la hora de implementar el personaje, es su capacidad de interactuar con su entorno: colisionar con el escenario, caer con la gravedad, detectar golpes del rival, etc. Todo esto dependerá de un motor de físicas que irá calculando todas estas interacciones a intervalos fijos. En Unity, se puede utilizar el motor de físicas que ofrece, que nos servirá para implementar las físicas 2D de nuestro juego.

Para el personaje, habrá que añadir un componente *Capsule Collider 2D* y un *Rigidbody* con la finalidad de poder simular su interacción física con el entorno. El Collider 2D es un componente que simula las colisiones dándole forma de cápsula al personaje. La razón para escoger una cápsula en vez de un cuadrado es que se adapta mejor a la forma del personaje. La figura 3.2 muestra una imagen con las propiedades del componente.

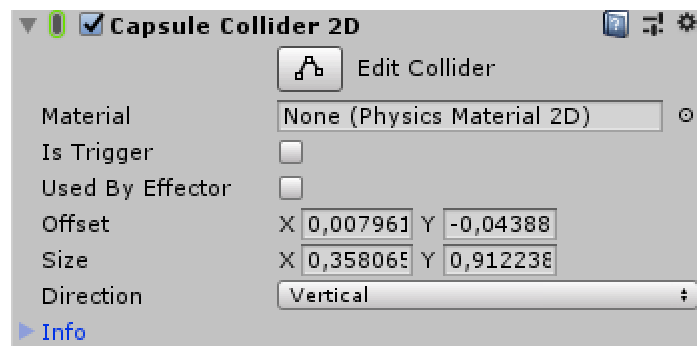


Figura 3.2: Propiedades del Collider 2D

El *Rigidbody 2D* es el componente que se encarga de hacer que el personaje sea un objeto físico como tal. Se encarga de darle masa y hacer que “exista” en el motor de física, en relación el resto de objetos. En este caso, tras darle una masa adecuada, es necesario también congelar la rotación en el eje Z, ya que nuestro personaje no va a rotar, físicamente hablando. Cualquier rotación que tenga será manejada usando animaciones. La figura 3.3 muestra las propiedades del Rigidbody 2D.

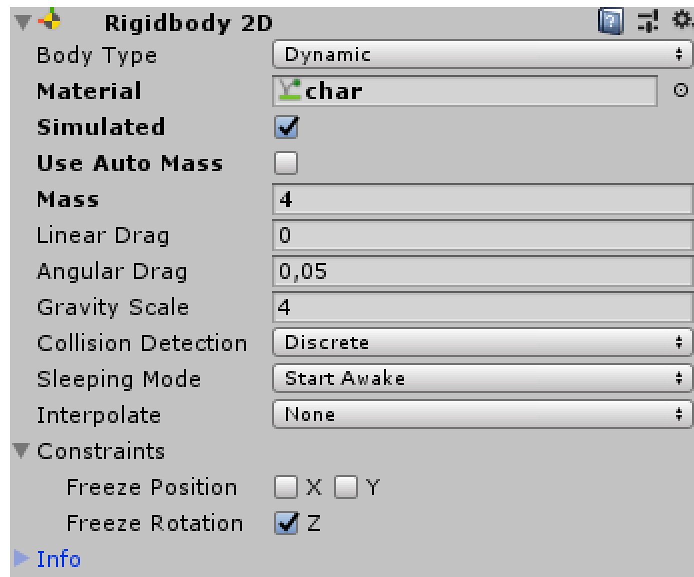


Figura 3.3: Propiedades del Rigidbody 2D

Con estos dos componentes, el personaje ya formará parte del motor de físicas y podrá interactuar con los demás objetos de la escena que también tengan algún tipo de física.

3.2.4 Script

Por último falta crear el componente que se encargará de ligarlo todo y hace que el personaje cobre vida: el script. Este componente contendrá toda la lógica necesaria para hacer que el jugador pueda controlar al personaje, y para que este pueda interactuar con todo lo que le rodea.

El scripting en Unity hace uso del lenguaje C#, y utiliza toda una API basada en parte en *Monogame*, que es la que maneja toda la lógica de los componentes. Cada componente es lo que se llama un *Monobehaviour* y creando una clase que herede de este puedes crear componentes nuevos en forma de script. Esto es lo que haremos para crear un componente "personaje" desde 0.

Los scripts funcionan mediante eventos y funciones, entre otros. Las funciones funcionan igual que en cualquier otro lenguaje de programación, mientras que los eventos son unas funciones especiales heredadas de Monobehaviour que serán llamadas por el motor de Unity en determinados momentos. Los eventos más comunes son:

- Start(): este evento se llama al crearse el objeto, y se puede utilizar para inicializar todos los datos necesarios del objeto.
- Update(): este evento se llama a cada fotograma, y sirve para programar toda la lógica principal del objeto.

A parte de estos, hay otros eventos menos comunes, además de eventos específicos para otros componentes, de los cuales se hablará más adelante. Por ahora vamos a analizar la lógica del personaje, es decir, el contenido de la función Update(), viendo las distintas secciones de código que se han ido implementando durante el desarrollo.

3.2.4.1 Movimiento

Lo primero que se debe hacer es capturar el Input, es decir, la entrada que usará el jugador para jugar al juego. Hay varias formas de hacer esto:

- Capturando la entrada de forma directa haciendo *polling* sobre el dispositivo, por ejemplo, comprobando si una tecla específica del teclado se está pulsando en un momento dado.
- Capturando la entrada de forma indirecta mediante acciones. Este método se basa en hacer *polling* de unas acciones predefinidas, y de forma paralela, asignar teclas a dichas acciones. Dicho de otra manera, lo que compruebas no es si se está pulsando una tecla, sino que le preguntas al sistema si se está realizando una acción. El sistema devolverá una respuesta en función de si las teclas asignadas a esa acción se están pulsando.

En general el segundo método es preferible al primero, dado que ofrece una flexibilidad que permite al jugador establecer los controles que más cómodos le sean para el juego, sin tener que modificar el código fuente. Es por ello que usaremos el sistema de acciones, y el resultado se muestra en el siguiente bloque de código:

```
mx = Input.GetAxis("Horizontal");
jmp = Input.GetButtonDown("Jump");
crch = Input.GetAxis("Crouch") > 0.4;
dash = Input.GetButtonDown("Guard/Dash");
grd = Input.GetButton("Guard/Dash");
pnch = Input.GetButtonDown("Punch");
```

```
kck = Input.GetButtonDown("Kick");
cnt = Input.GetButtonDown("Counter");
rng = Input.GetButtonDown("Ranged");
```

A cada una de las variables le almacenamos el valor retornado por el sistema de input, al inicio de la función Update(), de manera que en cada fotograma se actualice la entrada del jugador. Así logramos que el personaje conozca el input del jugador en todo momento.

A partir de aquí, el objetivo es ir implementando a medida cada parte de la jugabilidad y del funcionamiento del personaje. Se explicarán las secciones principales, incluyendo el código implementado de cada una.

Para el movimiento básico utilizaremos el feedback del Input que hemos almacenado en *mx* (movimiento en el eje X). Si el personaje está en el suelo y no está cargando en ninguna dirección, entonces comprobaremos en qué dirección está intentando moverse. En función de la dirección resultante, le aplicaremos una velocidad al Rigidbody del personaje de manera que se pueda mover, al mismo tiempo que enviaremos una señal al componente Animator para que realice las animaciones relacionadas con el movimiento.

Al mismo tiempo queremos manejar la posibilidad de saltar, que también será solo posible en caso de estar en el suelo y no estar cargando en ninguna dirección. De esta manera podemos controlar tanto el salto estático como el salto en movimiento, y el Animator se encargará de mostrar una animación u otra en función de ello. El bloque de código final se vería de la siguiente manera:

```
// movement
float vy = rb.velocity.y;

if (grounded && !dashing)
{
    canDash = true;
    if (mx > 0 && !grd)
    {
        dir = 0;
        if (!crouching)
        {
            rb.velocity = new Vector2(movSpeed, vy);
            anim.SetBool("Walking", true);
        }
    }
```

```
    else if (mx < 0 && !grd)
    {
        dir = 1;
        if (!crouching)
        {
            rb.velocity = new Vector2(-movSpeed, vy);
            anim.SetBool("Walking", true);
        }
    }
    else
    {
        rb.velocity = new Vector2(0, vy);
        anim.SetBool("Walking", false);
    }
    if (jmp)
    {
        anim.SetTrigger("Jump");
        if (!blockJmp)
        {
            rb.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);
            blockJmp = true;
        }
    }
}
```

La variable “dashing” es un booleano que estará a *true* cada vez que el personaje efectúe una carga, y a *false* en cualquier otro caso. “grounded” nos dará información sobre si el personaje está pisando el suelo en un momento dado o no. Para hacer esta última comprobación, utilizaremos un evento derivado del Collider2D asignado al personaje: OnCollisionEnter2D(). Este evento se ejecutará cada vez que el Collider2D del personaje entre en contacto con cualquier otro colisionador. En este caso, queremos comprobar que el contacto se ha realizado con el suelo, para poner “grounded” a *true*. De esta manera resulta el siguiente bloque de código:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.name == "Floor")
    {
        grounded = true;
        anim.SetBool("Grounded", true);
    }
}
```

De la misma manera que podemos comprobar que otro objeto entra en contacto con el personaje, también podemos hacerlo para cuando se acaba la colisión usando OnCollisionExit2D():

```
private void OnCollisionExit2D(Collision2D collision)
{
    if (collision.gameObject.name == "Floor")
    {
        grounded = false;
        anim.SetBool("Grounded", false);
    }
}
```

De esta manera sabremos en todo momento si el personaje está tocando el suelo o no, y servirá para bastantes acciones del mismo.

Volvamos al evento Update(). Queremos manejar la dirección en la que está mirando el personaje. Esto es importante porque el jugador debe recibir una respuesta visual de hacia dónde están mirando ambos personajes, ya que eso afectará a su forma de jugar. Para ello comprobaremos la dirección en la que se movió el personaje por última vez, y en caso de que cambie, girar la orientación del objeto:

```
// direction
float sx = transform.localScale.x;
float sy = transform.localScale.y;
if (lastDir != dir)
{
    transform.localScale = new Vector3(-sx, sy, 1.0f);
    lastDir = dir;
}
```

Los dos movimientos que quedan funcionan de manera similar a lo que hemos visto hasta ahora. Para agacharse, enviaremos un mensaje al Animator del personaje para indicar que queremos reproducir la animación de agachado:

```
// Crouching
if (crch)
{
    anim.SetBool("Crouch", true);
    crouching = true;
}
else
{
    anim.SetBool("Crouch", false);
    crouching = false;
}
```

El dash llevará algo más de trabajo, ya que a parte de las animaciones correspondientes, también hay que aplicarle una fuerza al Rigidbody del personaje para que se mueva en la dirección deseada. El código final quedaría así:

```
// dash
if (canDash && grd && mx != 0 && !dashing && dashReady && stamina >= dashStaminaCost)
{
    removeStamina(dashStaminaCost);
    rb.velocity = new Vector2(0, 0);
        if (mx > 0) rb.AddForce(new Vector2(dashSpeed, 0), ForceMode2D.Impulse);
    else rb.AddForce(new Vector2(-dashSpeed, 0), ForceMode2D.Impulse);

    setDashAnim(mx);
    anim.SetBool("Dashing", true);

    canDash = false;
    dashing = true;
    dashReady = false;
}
```

3.2.4.2 Acciones y timers

No todo es movimiento en este juego. También hay ciertas acciones de pueden realizar los personajes. En esta sección vamos a centrarnos en la acción principal de nuestro personaje: Atacar.

El bloque de código básico para tratar los ataques es el siguiente:

```
// attacks
if ((pnch || kck) && comboReady && !grd)
{
    if (pnch) anim.SetTrigger("Punch");
    else anim.SetTrigger("Kick");
    comboTime = 0;
    comboAct = true;
    comboReady = false;
    comboNum++;
}
```

Como puede observarse, el código para tratar los ataques es muy simple. Tan solo hay que darle orden al Animator de efectuar las animaciones adecuadas para cada ataque, y él se encargará internamente de manejar el resto del funcionamiento de

cada golpe. ¿Pero entonces para qué se necesitan las variables adicionales "comboTime", "comboAct", etc.?

La respuesta es sencilla. Si solo tuviéramos ese bloque de código, en cada fotograma se estaría comprobando si el jugador está intentando realizar un golpe, con lo cual los golpes del combo se ejecutarían demasiado rápido. La solución para esto es utilizar *timers*.

Los timers son variables de coma flotante que cuentan el tiempo que falta para poder volver a realizar una acción. Su funcionamiento es sencillo: al realizar una acción, la bloqueamos y ponemos su timer a 0, y en cada fotograma consultamos el tiempo que ha transcurrido desde el fotograma anterior (mediante la variable estática Time.deltaTime manejada por el motor) y se lo sumamos al timer, así cuando este llega a un valor preestablecido, liberamos el bloqueo de la acción de manera que se pueda volver a realizar. En el caso de los combos de ataque, el bloque de código que maneja su timer es el siguiente:

```
if (comboRestart)
{
    comboTime += Time.deltaTime;
    if (comboTime > comboRestartWindow)
    {
        comboRestart = false;
        comboReady = true;
    }
}

if (comboAct && comboReady)
{
    comboTime += Time.deltaTime;
    if (comboTime > comboResetWindow)
    {
        resetCombo();
    }
}

if (!comboReady)
{
    comboTime += Time.deltaTime;
    if (comboTime > comboReadyWindow)
    {
        comboReady = true;
        comboTime = 0;
    }
}
```

En este bloque de código, no solo controlamos el bloqueo de cada golpe del combo, sino que además controlamos que no pase demasiado tiempo entre un golpe y el siguiente del mismo combo. De esta manera, si pasa demasiado rato sin que el jugador ejecute una patada o un puñetazo, el combo se reiniciará al primer golpe.

Evidentemente, estos no son los únicos timers ni las únicas acciones que se pueden ejecutar. Tenemos el “ataque de rango” que no pertenece a ningún combo, con lo que utilizará su propio timer, como se muestra a continuación:

```
if (!rngReady)
{
    rngTimer += Time.deltaTime;
    if (rngTimer > rngTimerWindow)
    {
        rngTimer = 0f;
        rngReady = true;
    }
}

if (rng && rngReady && !grd && stamina > rangedStaminaCost)
{
    anim.SetTrigger("Ranged");
    rngReady = false;
    removeStamina(rangedStaminaCost);
}
```

En el caso de esta acción, hay que tener en cuenta que solo se puede realizar si el personaje tiene stamina suficiente, y una vez realizado hay que retirar la stamina gastada. Esta se recuperará haciendo uso de otro timer distinto, que hará que se recupere con el paso del tiempo:

```
staminaRefreshTimer += Time.deltaTime;
if (staminaRefreshTimer > staminaRefreshWindow)
{
    stamina++;
    staminaRefreshTimer = 0f;
}
if (stamina > maxStamina) stamina = maxStamina;
```

Como puede observarse, el uso de timers es imprescindible para poder controlar el flujo de control del personaje, y de cualquier objeto que realice acciones a lo largo del tiempo. Otros timers que se están utilizando para la lógica del personaje son:

- Timer de contraataque: Para controlar el tiempo que pasa entre que se bloquea un golpe y se pulsa la acción de contraataque, de manera que si se tarda demasiado en pulsar, el contraataque se pierde.
- Timer de Dash: Para controlar que no puedas hacer demasiados dashes seguidos.
- Timer de bloqueo roto: Una vez le rompe el bloqueo su rival, hay una pequeña ventana de tiempo en la cual el personaje no puede realizar ninguna acción. Este timer se usa para controlarla.
- Timer de invencibilidad: Cuando le hacen knockback a un personaje, hay una pequeña ventana de tiempo en la cual este es invencible, para evitar que alguien abuse de los tiempos de recuperación de su rival.

3.2.5 Detalles finales

En la sección anterior se ha comentado que al realizar un ataque, se envía una señal al Animator, y este internamente se encarga de la lógica y la animación del ataque. Pero evidentemente, esto no lo hará por arte de magia. Es necesario implementar scripts de animación. Estos scripts no heredan de *MonoBehaviour*, sino de *StateMachineBehaviour*, con lo cual no funcionan como componentes de objetos, y por lo tanto no contienen los eventos Update() o Start().

Estos scripts se asignan a estados de la máquina de estados, y por lo tanto los eventos que emiten hacen referencia a cuando se entra o se sale del estado al que están asignados, por ejemplo. Sabiendo esto, podemos definir un evento para los estados de cada ataque que se ejecute al inicio del estado, y que lance un nuevo evento para indicar que se está realizando el ataque, y las características del mismo:

```
public override void OnStateEnter(Animator animator, AnimatorStateInfo
stateInfo, int layerIndex)
{
        Dictionary<string, object> parameters = new Dictionary<string,
object>();
    parameters.Add("type", type);
    parameters.Add("dmg", dmg);
    parameters.Add("source", animator.gameObject);
    parameters.Add("finisher", finisher);
    animator.gameObject.SendMessage("doStrike", parameters);
    if (endCombo)
```

```
        animator.gameObject.SendMessage("resetComboMessage");
}
```

En este bloque de código se puede observar algo que no hemos visto hasta ahora. La función SendMessage() dispara un evento personalizado, igual que los que Unity manda a todos los componentes (Update(), Start(), etc.), pero con la diferencia que la función que se llamará será alguna que hayamos definido nosotros mismos. En este caso se llamará la función doStrike() con “parameters” como parámetro de entrada de la llamada a función, la cual está definida en el script del personaje:

```
private void doStrike(Dictionary<string, object> p)
{
    if (target == null || !targetInRange) return;

    // stamina modifier
    int? dmg = p["dmg"] as int?;
    dmg = 5 + dmg * stamina / maxStamina;
    p["dmg"] = dmg;

    // send damage to target
    target.SendMessage("receiveDamage", p);
}
```

Esta función calculará el daño final utilizando la fórmula del apartado 2.3 y enviará un nuevo evento al personaje receptor del golpe para indicarle que está recibiendo un golpe, junto con los atributos del mismo. Esta función se encargará de comprobar si el daño del golpe se efectuará finalmente, en función del estado del personaje receptor:

```
private void receiveDamage(Dictionary<string, object> p)
{
    GameObject source = p["source"] as GameObject;
    if (invincible || finished)
    {
        if (source.tag == "projectile")
            Destroy(source);
        return;
    }

    string type = p["type"] as string;
    int dmg = (int)(p["dmg"] as int?);

    lastHit = p;
    int dmgDir = 0;
    if (source.transform.position.x < transform.position.x) dmgDir = 1;
    if (dmgDir == dir && (guardHigh || guardLow))
    {
```

```
        blockedLastHit = true;
        if (source.tag != "projectile")
        {
            if (guardBroken())
                return;
            source.SendMessage("blocked", dmg);
            if (checkCounter(source, type) && guardHigh)
            {
                canCounter = true;
            }
        }
        if (!crouching)
            removeStamina(dmg);
        else
            removeStamina(dmg / 2);
    }
    else
    {
        blockedLastHit = false;
        doDamage();
    }

    if (source.tag == "projectile")
        Destroy(source);
}
```

A su vez, esta función le devolverá al personaje atacante una señal en caso de ser bloqueado, y se encargará de activar los timers necesarios en caso de que se pueda contraatacar o de que el bloqueo sea roto. Se podría resumir el flujo de acontecimientos de la siguiente manera:

- El jugador pulsa acción de ataque.
- El script de personaje detecta la pulsación y envía señal al Animator de que inicie la animación de golpe
- El Animator detecta la señal de golpe e intenta iniciar la animación requerida.
- El estado del Animator detecta que se está ejecutando su animación y ejecuta el código que le ha sido asignado, enviando disparando un evento de vuelta al personaje atacante.
- El evento es recogido por el personaje, que calcula el daño final del golpe y se lo envía al receptor del golpe en otro evento.
- Este último evento es recibido por el script del personaje receptor del golpe, y este decide si el daño ocurrirá de verdad o no, y reaccionará de manera acorde.

Como puede observarse, el proceso es muy modular y autónomo, de manera que si en algún momento hay que modificar alguna parte del mismo, solo hay que cambiar esa parte sin preocuparse de las demás, o de cómo les pueda afectar.

3.3 Otros objetos

Una vez acabado el personaje, es necesario poblar la escena con el resto de objetos que podrían ser necesarios o interesantes de añadir. Haremos un pequeño recorrido por estos.

3.3.1 Cámara y limitadores de escenario

Aparte del personaje, hay algunos objetos que son completamente necesarios en este y en cualquier juego. Uno de ellos es la *cámara*. Este objeto es el encargado de "visualizar" la escena.

Las cámaras hacen *render* de la escena para poder procesar todo el apartado gráfico de lo que "existe" en la misma. De esta manera, tanto el personaje como el escenario pasarán de ser simples datos a ser una imagen visible en el monitor, algo que es evidentemente imprescindible en un juego de este tipo (si no podemos ver el juego, no podemos jugarlo).

En un juego 2D, queremos que la proyección de la cámara sea ortográfica. Esto significa que el espacio proyectado se trata como una estructura cúbica, de manera que no se genere una sensación de profundidad en el juego (lo cual es obvio si consideramos que un juego 2D son imágenes planas superpuestas unas sobre otras). Si se tratara de un juego 3D, sí que nos interesaría que la proyección de la cámara fuera perspectiva, ya que en ese caso sí que queremos esa sensación de profundidad, pero en nuestro caso no es necesario ni recomendable.

Los otros objetos imprescindibles son los limitadores del escenario: paredes y suelo. No queremos que nuestro personaje pueda irse fuera de los límites del escenario, o que se caiga hacia abajo. Para ello crearemos unos objetos invisibles que tendrán un componente Collider2D cada uno, y con los cuales delimitamos el escenario. El

suelo en particular nos servirá para que el personaje sepa si está “grounded” a la hora de ejecutar su lógica.

3.3.2 Fondo y UI

A continuación se explicarán otros objetos que no son completamente imprescindibles para jugar, pero que son altamente recomendables de cara al usuario final del videojuego, y que forman parte de la definición de cualquier juego que se valore: el escenario y la *User Interface* (UI).

El escenario generalmente es lo primero que el jugador verá cuando empiece a jugar. Es algo que ocupa toda la pantalla, mientras que los personajes solo requieren una pequeña fracción de esta. Es por ello que es importante hacer escenarios amigables para la vista y que no distraigan al jugador, al mismo tiempo que generan una cierta inmersión en la partida. En el caso de este juego, el escenario es un fondo animado (un fichero mp4) que se reproduce de fondo para acompañar a los personajes durante el combate.

Por otro lado tenemos la *User Interface* (UI), o interfaz de usuario. Esta es el conjunto de la información en pantalla que puede resultar útil para el jugador, y que generalmente se superpone a todo el contenido de la partida.

Mientras que se puede jugar sin la información proporcionada por dicha UI, es altamente recomendable que se implemente como soporte al jugador. En este caso, consideraremos que la UI esté comprendida por las barras de vida y de stamina de ambos personajes, de manera que el jugador pueda tener información sobre la progresión del combate. Es una UI bastante minimalista, aunque se le podrían añadir más funciones, como número de combates ganados, tiempo de combate, o incluso contador de combos y daño. Todo dato añadido es información adicional para el jugador, pero hay que ir con cuidado de no llenar la UI con demasiada información, ya que puede ser contraproducente por las siguientes razones:

- Demasiada información genera distracción, lo cual en un juego como este, donde la concentración y la reacción son vitales, es fatal.

- A más llena la UI, menos espacio en pantalla para ver el juego en sí, lo cual dificulta la visibilidad al jugador.
- Si hay sobrecarga de información, el jugador puede llegar a dejar de procesar parte de esta.

Es por ello que hay que buscar un buen balance a la hora de diseñar una UI. Deben añadirse los elementos que puedan ayudar al jugador, pero sin añadir demasiados, ya que podrían convertirse en obstáculos para el mismo.

3.4 IA del bot

En el apartado 2.5 ya se habló sobre el funcionamiento del aprendizaje reforzado, algunos de sus algoritmos y estrategias a seguir de cara a las pruebas. En este apartado se explicará el proceso de implementación de la IA en Unity, haciendo uso de la API "ml-agents" que proporciona de cara a implementar agentes de aprendizaje automático en los juegos.

3.4.1 Objetos de la API

La API de agentes de aprendizaje de Unity introduce tres tipos de objeto, todos imprescindibles para el buen funcionamiento de la IA y su aprendizaje:

- *Agent*: El objeto básico de todo aprendizaje automático. Un agente es una entidad que observa su entorno y envía la información obtenida a un sistema que procesa dicha información y toma decisiones, tras las cuales el agente las recibe y las procesa para actuar en consecuencia. Puede haber tantos agentes como sea necesario en una misma escena.
- *Brain*: Es la entidad que recibirá la información del entorno obtenida por los agentes, y les devolverá una serie de decisiones en función de unos parámetros que se comentarán más adelante. Puede haber tantos Brains como sea necesario en una misma escena.
- *Academy*: Es el objeto que administra todo el sistema de aprendizaje, controlando todos los Brains y Agentes de la escena. Contiene la configuración global del aprendizaje de sus agentes, y se pueden scriptear

algunas partes de esta, aunque generalmente no suele ser necesario (solo para aprendizajes muy específicos). Dada su finalidad, únicamente debe haber una academia por escena.

Como mínimo debe haber un objeto de cada en cualquier escena en la que queramos usar un agente de aprendizaje, ya que todos tienen su papel a desarrollar dentro del sistema.

3.4.2 Modos del Brain

Los Brains tienen 3 modos de funcionamiento distintos, y cada uno tiene su finalidad. A continuación se explicará cada uno de ellos:

- *Internal*: en esta modalidad, el Brain utilizará datos preprocesados para tomar las decisiones en función de las observaciones del agente. Generalmente este es el modo que nos interesa una vez la IA está suficientemente entrenada, y es la que se usará en partidas contra bots cuando no esté en modo entrenamiento.
- *External*: en esta modalidad, el Brain se conectará a un proceso separado de Unity a través de un socket del PC. Este proceso es un programa en python que implementa una red neuronal que recibe la información del agente, la procesa y toma decisiones que le envía de vuelta al agente. Como bien dice el nombre, el Brain es externo a Unity. Esta es la modalidad que se debe utilizar para entrenar las IAs, dado que la red neuronal aplicará el algoritmo implementado (PPO en este caso) para entrenar los agentes.
- *Player*: en esta modalidad, el Brain se desactiva y permite establecer teclas del teclado que sirvan para simular las decisiones que este tomaría. Este método es útil para probar si el agente está recibiendo las órdenes del Brain de forma correcta, así como asegurarse de que las respuestas del mismo agente son las esperadas a partir de las decisiones que toma el Brain.

3.4.3 El agente

El plato principal de esta sección es el agente de aprendizaje, que es donde realmente se va a implementar la estrategia de aprendizaje, y es el actor principal.

Durante esta sección se pondrá ejemplos para cada explicación usando la implementación de la tercera estrategia del diseño.

Para desarrollar el comportamiento del agente habrá que implementar una clase que hereda de *Agent* en un script que irá enganchado al propio agente. Este agente ha de ser rellenado con diversas funciones que se explicarán en las próximas secciones.

3.4.3.1 Observaciones

El primer paso es establecer el vector de observaciones del agente, que es la información que le pasará al Brain para que se procesen las acciones consecuentes. Para ello habrá que implementar una función CollectObservations() en el script del agente que será la encargada de recopilar toda la información relevante para el aprendizaje, y enviarla al Brain mediante la función AddVectorObs(). Esta función es una implementación directa de las observaciones que se han establecido en el apartado 2.5 del documento, y en el caso de la tercera estrategia, el código de dicha función sería el siguiente:

```
public override void CollectObservations()
{
    // Calculate relative position
    Vector3 relativePosition = target.transform.position -
this.transform.position;

    // Relative position to target
    AddVectorObs(relativePosition.x / 10f);
    AddVectorObs(relativePosition.y / 10f);

    // Agent velocity
    AddVectorObs(rb.velocity.x / 10f);
    AddVectorObs(rb.velocity.y / 10f);

    // Target velocity
    AddVectorObs(target.GetComponent<Rigidbody2D>().velocity.x / 10f);
    AddVectorObs(target.GetComponent<Rigidbody2D>().velocity.y / 10f);

    // Directions
    AddVectorObs(this.GetComponent<Character>().dir);
    AddVectorObs(target.GetComponent<Character>().dir);

    // Stats
AddVectorObs(((float)this.GetComponent<Character>().getStamina())/100f);

    // Animations
```

```
AddVectorObs(this.GetComponent<Animator>().GetCurrentAnimatorStateInfo(0).s
hortNameHash/100000000f);

AddVectorObs(target.GetComponent<Animator>().GetCurrentAnimatorStateInfo(0)
.shortNameHash/100000000f);

    // Projectile incoming
    int incoming = 0;
    foreach (GameObject g in
GameObject.FindGameObjectsWithTag("projectile"))
    {
        if (g.transform.position.y < this.transform.position.y + 2 &&
g.transform.position.y > this.transform.position.y - 2)
        {
            if (g.GetComponent<Rigidbody2D>().velocity.x > 0f &&
g.transform.position.x < this.transform.position.x) incoming++;
            if (g.GetComponent<Rigidbody2D>().velocity.x < 0f &&
g.transform.position.x < this.transform.position.x) incoming++;
        }
    }
    AddVectorObs(incoming/5f);
}
```

Como puede observarse, implementar la función de observaciones es relativamente sencillo, y permite bastante flexibilidad a la hora de probar distintas estrategias, lo cual facilita la fase de pruebas y experimentación.

3.4.3.2 Acciones y recompensas

Una vez hemos establecido las observaciones que se enviarán al Brain, es el momento de implementar la recepción de las decisiones de este, así como establecer las recompensas y penalizaciones en función del estado actual de la partida. Para realizar estas dos acciones, se debe implementar la función AgentAction() en el script del agente, que recibirá por parámetro de entrada las acciones decididas por el Brain. En esta función podremos utilizar las acciones enviadas por el Brain como sea conveniente, además de que podremos enviar recompensas y penalizaciones al mismo mediante la función AddReward(float v), donde valores positivos de “v” significarán recompensas mientras que los negativos representarán penalizaciones. A continuación se muestra la implementación de esta función para la tercera estrategia:

```
public override void AgentAction(float[] vectorAction, string textAction)
{
    // Rewards
       float  distanceToTarget  =  Vector3.Distance(this.transform.position,
target.transform.position);
    int hp, thp, stamina, tstamina;
    hp = this.GetComponent<Character>().getHP();
    thp = target.GetComponent<Character>().getHP();
    stamina = this.GetComponent<Character>().getStamina();
    tstamina = target.GetComponent<Character>().getStamina();

    // Stats rewards
    AddReward((prevTHP-thp)*0.01f);
    AddReward((hp-prevHP)*0.01f);
    AddReward((stamina - prevStamina)*0.001f);

    AddReward(-0.001f);

    // Death
    if (thp <= 0)
    {
        AddReward(1.0f);
        prevHP = this.GetComponent<Character>().maxHp;
        prevTHP = target.GetComponent<Character>().maxHp;
        prevStamina = this.GetComponent<Character>().maxStamina;
        prevTStamina = target.GetComponent<Character>().maxStamina;
        hp = prevHP;
        thp = prevTHP;
        stamina = prevStamina;
        tstamina = prevTStamina;
        Done();
    }
    else if (hp <= 0)
    {
        AddReward(-1.0f);
        prevHP = this.GetComponent<Character>().maxHp;
        prevTHP = target.GetComponent<Character>().maxHp;
        prevStamina = this.GetComponent<Character>().maxStamina;
        prevTStamina = target.GetComponent<Character>().maxStamina;
        hp = prevHP;
        thp = prevTHP;
        stamina = prevStamina;
        tstamina = prevTStamina;
        Done();
    }

    previousDistance = distanceToTarget;
    prevHP = hp;
    prevTHP = thp;
    prevStamina = stamina;
    prevTStamina = tstamina;
}
```

3.4.3.3 Adaptando el Input del personaje al agente

Ahora tan solo falta ver cómo afectan las decisiones del Brain al propio personaje rival, ya que lo único que recibimos de este es un vector con una lista de acciones, y eso no hará funcionar al personaje automáticamente.

Para realizar esta tarea, se decidió simular un sistema de Input desde el código del agente, de manera que el script del personaje pueda acceder a él como si del Input del jugador se tratase. Para esto será necesario almacenar las acciones del Brain en cada actualización del agente, y para ello usaremos la misma función que hemos creado en el apartado anterior: AgentAction(). Almacenaremos la información de las acciones en unos arrays que nos permitirán consultar dichas acciones en todo momento. Así pues, añadiremos las líneas de código siguientes al final de la función:

```
// Actions, size = 8
input["Horizontal"] = vectorAction[0]; // Movement
input["Crouch"] = vectorAction[1] > 0f; // Crouch
input["Guard/Dash"] = vectorAction[2] > 0f; // Guard/Dash
input["Counter"] = vectorAction[3] > 0f; // Counter
input["Jump"] = vectorAction[4] > 0f; // Jump
input["Punch"] = vectorAction[5] > 0f; // Punch
input["Kick"] = vectorAction[6] > 0f; // Kick
input["Ranged"] = vectorAction[7] > 0f; // Ranged

inputdown["Horizontal"] = false; // Movement
inputdown["Crouch"] = GetButton("Crouch") && !pcrouch; // Crouch
inputdown["Guard/Dash"] = GetButton("Guard/Dash") && !pguard; // Guard/Dash
inputdown["Counter"] = GetButton("Counter") && !pcounter; // Counter
inputdown["Jump"] = GetButton("Jump") && !pjump; // Jump
inputdown["Punch"] = GetButton("Punch") && !ppunch; // Punch
inputdown["Kick"] = GetButton("Kick") && !pkick; // Kick
inputdown["Ranged"] = GetButton("Ranged") && !pranged; // Ranged

inputup["Horizontal"] = false; // Movement
inputup["Crouch"] = !GetButton("Crouch") && pcrouch; // Crouch
inputup["Guard/Dash"] = !GetButton("Guard/Dash") && pguard; // Guard/Dash
inputup["Counter"] = !GetButton("Counter") && pcounter; // Counter
inputup["Jump"] = !GetButton("Jump") && pjump; // Jump
inputup["Punch"] = !GetButton("Punch") && ppunch; // Punch
inputup["Kick"] = !GetButton("Kick") && pkick; // Kick
inputup["Ranged"] = !GetButton("Ranged") && pranged; // Ranged

pcrouch = GetButton("Crouch");
pguard = GetButton("Guard/Dash");
pcounter = GetButton("Counter");
pjump = GetButton("Jump");
ppunch = GetButton("Punch");
```

```
pkick = GetButton("Kick");
pranged = GetButton("Ranged");
```

Una vez tenemos toda la información almacenada, necesitamos un conjunto de funciones que simulen el comportamiento de los métodos de consulta de la clase Input. Para ello nos haremos nuestras propias funciones para consultar el Input del agente en su script. Dichas funciones son bastante simples:

```
public float GetAxis(string s)
{
    return (float)(input[s] as float?);
}

public bool GetButton(string s)
{
    return (bool)(input[s] as bool?);
}

public bool GetButtonDown(string s)
{
    return (bool)(inputdown[s] as bool?);
}

public bool GetButtonUp(string s)
{
    return (bool)(inputup[s] as bool?);
}
```

Finalmente, modificamos el script de personaje, para permitir que el Input pueda ser recogido del agente en vez del jugador, en caso de que el personaje sea controlado por IA:

```
if (playerControlled)
{
    mx = Input.GetAxis("Horizontal");
    jmp = Input.GetButtonDown("Jump");
    crch = Input.GetAxis("Crouch") > 0.4;
    dash = Input.GetButtonDown("Guard/Dash");
    grd = Input.GetButton("Guard/Dash");
    pnch = Input.GetButtonDown("Punch");
    kck = Input.GetButtonDown("Kick");
    cnt = Input.GetButtonDown("Counter");
    rng = Input.GetButtonDown("Ranged");
    if (Input.GetButtonUp("Guard/Dash") || mx == 0)
    {
        dashReady = true;
    }
}
else
{
    mx = agent.GetAxis("Horizontal");
```

```
    jmp = agent.GetButtonDown("Jump");
    crch = agent.GetButton("Crouch");
    dash = agent.GetButtonDown("Guard/Dash");
    grd = agent.GetButton("Guard/Dash");
    pnch = agent.GetButtonDown("Punch");
    kck = agent.GetButtonDown("Kick");
    cnt = agent.GetButtonDown("Counter");
    rng = agent.GetButtonDown("Ranged");
    if (agent.GetButtonUp("Guard/Dash") || mx == 0)
    {
        dashReady = true;
    }
}
```

Con este método, no solo hemos conseguido construir un personaje controlado por un bot de forma sencilla y rápida, sino que realmente funcionará como si fuera un segundo jugador, con la diferencia de que quien lo controle no será un jugador real sino un agente de aprendizaje. Además esto hace que sea sencillo cambiar el comportamiento de un personaje, ya que tenemos una variable booleana que determina si dicho personaje será controlado por un jugador o por la IA.

4. Descripción de resultados

Una vez implementado el juego y el bot, es necesario realizar pruebas para ver si funciona correctamente. Para ello se diseñaron tres estrategias en el apartado 2.5, de manera que tuviéramos distintos escenarios sobre los que probar el aprendizaje del agente.

El proceso de experimentación para cada estrategia es el siguiente:

- Se implementan los parámetros de la estrategia en el código del agente.
- Se efectúan 10 horas de entrenamiento jugador vs bot para cada estrategia.
- Se calcula la mejora media de la recompensa a cada hora de entrenamiento
- A partir de estos datos, además de los datos obtenidos directamente durante las partidas (jugando), intentar establecer cuál de las tres estrategias es más efectiva para el aprendizaje del agente.

Así pues, para cada estrategia se establecerá una tabla de recompensas para cada hora de entrenamiento, además de un análisis detallado del comportamiento del bot en combate, y un razonamiento que explique las recompensas resultantes.

Las recompensas se calculan en fragmentos cada 1000 actualizaciones, que es equivalente a unos 15 segundos de entrenamiento. La recompensa media en una hora de entrenamiento es la media de las recompensas de todos los fragmentos que suceden en esa hora (aproximada a las décimas). En la tabla final se incluirá tanto la recompensa media para cada hora, como la diferencia respecto a la hora anterior, de manera que se pueda observar el progreso del entrenamiento, y confirmar si realmente el bot está mejorando, es decir, si las diferencias de recompensa tienden a incrementar.

4.1 Estrategia 1

4.1.1 Tabla de recompensas

La tabla de recompensas de esta estrategia es como se muestra a continuación:

Hora de entrenamiento	Recompensa media	Diferencia
0	-40.3	-----
1	-56.6	-16.3
2	-64.2	-7.6
3	-64.4	-0.2
4	-64.5	-0.1
5	-64.4	+0.1
6	-64.5	-0.1
7	-64.5	-----
8	-64.3	+0.2
9	-64.4	-0.1
10	-64.4	-----

4.1.2 Análisis

En la tabla de puede observar un comportamiento algo inesperado en la evolución de las recompensas. A partir de la tercera hora, las recompensas se quedan bloqueadas en un número muy bajo, y no parece que vaya a mejorar fácilmente.

En realidad es bastante sencillo ver por qué ocurre eso, dado el diseño de la estrategia. La razón de base, es que la estrategia es incorrecta: está mal diseñada. Es cierto que los parámetros de observación son extensos y complejos, y que eso haría que el aprendizaje sea muy lento, pero también muy detallado, aunque esa no

es la cuestión en este caso. El problema viene en las recompensas y penalizaciones.

El aprendizaje funciona mediante el análisis de las recompensas y penalizaciones en cada estado, por lo que si podemos forzar al bot a recibir recompensas o penalizaciones en los momentos que nos interesen, podemos romper completamente su aprendizaje. El diseño de esta estrategia tiene un agujero enorme en el cual podemos forzar estas recompensas y penalizaciones como queramos: la stamina.

Esta estrategia aplica recompensa y penalización cuando la stamina del rival sube o baja. Pero evidentemente, que suba o baje depende en gran medida de las acciones del rival. Entonces pongamos una situación en la que el jugador, a sabiendas de esto, decide gastar stamina cada vez que el bot está realizando alguna acción potencialmente buena, y en cambio decide recuperar stamina que el bot está realizando alguna acción potencialmente mala. El resultado de estas acciones será catastrófico para el bot, ya que creerá que acciones malas son buenas, debido a las recompensas que recibirá por culpa de su rival perdiendo stamina, mientras que creerá que acciones buenas son malas, por la misma razón.

Por lo tanto, considerando este problema de diseño, además de la inmensa cantidad de tiempo que haría falta para que el bot aprenda con los parámetros de observación utilizados, esta estrategia es incorrecta, inviable, y será descartada para el futuro.

4.2 Estrategia 2

4.2.1 Tabla de recompensas

La tabla de recompensas de esta estrategia es como se muestra a continuación:

Hora de entrenamiento	Recompensa media	Diferencia
0	-22.5	-----
1	-28.3	-5.8
2	-15.2	+13.1
3	-18.3	-3.1
4	-20.2	-1.9
5	-12.5	+7.7
6	-15.7	-3.2
7	-25.5	-9.8
8	-24.3	+1.2
9	-22.6	+1.7
10	-30.8	-8.2

4.2.2 Análisis

A diferencia de la estrategia 1, en esta el aprendizaje ya no se queda estancado, ya que las recompensas dependen enteramente de las acciones del bot y no se pueden manipular.

En este caso nos encontramos con una gran variancia en las diferencias entre recompensas a cada hora de entrenamiento. No parece que incrementen o decrementen de manera regular, y tampoco siguen un patrón fijo. Esto se puede explicar mediante las observaciones realizadas durante las partidas contra el bot.

Las recompensas y penalizaciones recibidas se notaban bastante rápido, ya que los cambios de política de decisión del agente cambiaban bastante bruscamente en cada ciclo. Por ejemplo, enseguida se daba cuenta de que acercarse al rival e intentar golpearlo sin control podía dar buenos resultados ya que hacía bastante daño con ese método en sus primeras partidas, y por lo tanto intentaba explotarlo.

Pero claro, su rival también cambia su manera de jugar, ya que es un jugador real, y aprende igual que él. De manera que en cuanto se veía al bot en modo ofensivo, simplemente había que cambiar a una estrategia más defensiva basada en bloqueos y contraataques, los cuales hacían al bot perder la partida. Y dada la simplificación en los parámetros de observación de esta estrategia, el bot agente no tenía forma de entender por qué su política efectiva hasta ese punto dejaba de serlo, hasta que volvía a encontrar beneficio en atacar sin control, repitiéndose así el ciclo. Esto explica el hecho de que las recompensas medias en la tabla suban y bajen de esta manera.

Se podría decir que esta estrategia es simple y eficaz de cara a jugadores con poca experiencia jugando a este tipo de juegos, ya que su aprendizaje es rápido pero no muy detallado, con lo cual ofrece un desafío inmediato a sus rivales, pero a la larga deja de ser tan efectivo ya que a la que el jugador mejora suficiente, le es muy sencillo contrarrestar las estrategias del bot.

4.3 Estrategia 3

4.3.1 Tabla de recompensas

La tabla de recompensas de esta estrategia es como se muestra a continuación:

Hora de entrenamiento	Recompensa media	Diferencia
0	-38.6	-----
1	-42.2	-3.6
2	-43.1	-0.9
3	-41.2	+1.9
4	-39.8	+1.4
5	-37.9	+1.9
6	-34.5	+3.4
7	-33.2	+1.3
8	-31.5	+1.7
9	-28.2	+3.3
10	-27.5	+0.7

4.3.2 Análisis

En esta última estrategia se puede observar unos resultados mucho más regulares que en las anteriores. La recompensa media empieza a mejorar de manera constante a partir de la tercera hora, con algunos picos en algunos puntos.

Empecemos por describir la forma de jugar del bot usando esta estrategia. Durante las primeras dos horas no cambió mucho respecto a las otras estrategias, simplemente intentaba calibrar sus acciones de forma aleatoria independientemente de sus observaciones. Aunque poco a poco se iba viendo algo de progreso.

Por ejemplo, después de estar bastante rato disparándole con ataques de rando de manera indefinida, aprendió que debía bloquearlos para no perder vida. Al cabo de unas cuantas horas aprendió que mejor que bloquear era esquivarlos, ya que eso le permitía mantener la stamina, mientras que bloquear se la hacía bajar. Al final del entrenamiento era capaz de esquivar prácticamente todos los proyectiles que le lanzaba.

Lo importante es observar que esta estrategia genera un plan de entrenamiento a medio plazo que permite al bot ir mejorando de forma constante y detallada. Esto se debe a que los parámetros de observación tienen en cuenta gran parte de parámetros importantes, pero sin sobrecargar al agente con un área de observación demasiado grande como para recorrerla en un tiempo computable. Además las recompensas y penalizaciones se han simplificado lo suficiente como para que sean representativas de los objetivos de un jugador real, al mismo tiempo que dependen únicamente de las acciones del bot, de forma que no se puedan manipular de manera externa (igual que ocurría en la estrategia 1).

Si extrapolamos los datos obtenidos, y consideramos que el rival del bot también irá jugando mejor a medida que vea que el agente aprende, haciendo que el aprendizaje se ralentice un poco a medida que avanza el tiempo, podríamos aproximar que a partir de las 150 horas, el bot empezará a ofrecer un desafío a la altura, y en poco menos del doble de tiempo podría empezar a ganar partidas de forma regular. Esto convierte esta estrategia en la más adecuada de las tres que se han estudiado, a pesar de que probablemente se puede encontrar una estrategía todavía mejor que esta, que permita al agente aprender más rápido y con el mismo nivel de detalle.

5. Gestión del proyecto

5.1 Metodología

Considerando que el trabajo es extenso y el tiempo de desarrollo es limitado, fue necesario establecer unas metas que se debían cumplir, para asegurar que el proyecto avanzaba de forma continua y eficiente. Además serviría como guía para saber si se va bien de tiempo respecto al calendario establecido, de manera que se pueda avanzar trabajo futuro, y así poder extender la parte final del trabajo realizando más pruebas o implementando nuevas ideas que surjan durante el desarrollo del mismo. Aunque la planificación cambió, la metodología utilizada es la misma que al principio, aunque se modificaron los objetivos.

5.1.1 Seguimiento

Se procura seguir las pautas y las fechas límites autoimpuestas. Siempre que haya progresos notables, se informará al director del proyecto, y de la misma manera, se le consultarán dudas que puedan estancar el desarrollo del mismo, para que éste le ofrezca algún tipo de guía o ayuda al respecto.

Al mismo tiempo, para realizar las pruebas del juego, se necesitan jugadores, para ello se pedirá la colaboración de compañeros de facultad para que prueben el juego y den *feedback* al respecto. Toda crítica constructiva se tendrá en cuenta para la mejora del trabajo final.

Finalmente, no se ha podido seguir la planificación preestablecida, pero el seguimiento con el director del proyecto se ha realizado igual, como se ha establecido previa realización del trabajo.

5.1.2 Herramientas de desarrollo

Finalmente, es necesario establecer las herramientas que se usarán durante el desarrollo del proyecto.

Primero de todo, el juego se implementará en Unity. Este motor gráfico es muy adecuado como base para desarrollar el juego, por diversas razones:

- Te ofrece las herramientas básicas para desarrollar tu propio juego sin tener que dedicarle mucho tiempo a la estructura base del motor, en el apartado gráfico o sonoro, entre otros.
- Ofrece opciones muy extensas a la hora de implementar un juego, desde *scripting* para el comportamiento de los objetos del juego, hasta inclusión de *plugins* externos para añadir funcionalidades adicionales.
- Tiene una documentación muy elaborada.
- Ofrece una API de *Machine Learning Agents*[16], es decir, agentes de aprendizaje automático, lo cual es totalmente idóneo para la implementación de las técnicas de aprendizaje por refuerzo necesarias para el proyecto.

Además, se planeó utilizar un repositorio privado en Github para mantener el proyecto en la nube, pero finalmente se han usado los repositorios privados del propio Unity, por cuestiones de funcionalidad y comodidad, ya que en caso de que fuera necesario trabajar en ordenadores distintos a lo largo del desarrollo del proyecto, además de que sirve como copia de seguridad en caso de necesidad, ya que se puede recuperar cualquier versión subida al repositorio. Así si en algún momento se comete algún error grave, se puede restaurar una versión anterior si es necesario. Además, el director del proyecto podrá acceder a dicho repositorio para observar el avance del trabajo en todo momento.

5.2 Planificación

El desarrollo del proyecto tenía una duración aproximada de 4 meses, del 19 de febrero al 29 de junio de 2018. Esta estimación no se pudo completar por razones externas al desarrollo del proyecto, con lo cual se presentará la planificación relativa a la duración final del mismo.

[16] Referencias y documentación en [3]

5.2.1 Hito inicial

Esta es la fase inicial del proyecto, y consta de:

- Definición del alcance, contextualización y bibliografía
- Planificación temporal
- Gestión económica y sostenibilidad

5.2.2 Análisis y diseño

Inicialmente debemos analizar los detalles relevantes para el proyecto, y establecer el diseño del mismo, previo a la implementación.

El análisis inicial del proyecto incluye cuestiones como los objetivos del proyecto (ya estudiados como parte del hito inicial), los requisitos del mismo (*hardware* y *software*, por ejemplo, entre otras cosas), y las funcionalidades necesarias. También se debe realizar un posicionamiento del proyecto con respecto al estado del arte, para determinar la situación del mismo en el mundo actual.

Por lo que respecta al diseño, se tiene que determinar primero todos los aspectos necesarios para el desarrollo del juego (jugabilidad, contenido, extensión, etc), así como las características de la IA que se va a desarrollar para el bot, que requerirá bastante trabajo teórico previo a la propia implementación del programa.

Para todo esto se aplicarán tanto conocimientos estudiados a lo largo de la especialidad del grado, como conocimientos aprendidos por cuenta propia, y añadiendo nuevos conceptos que se estudiarán y trabajarán con el objetivo de ser aplicados en este proyecto.

5.2.3 Descripción de las tareas

Las tareas relevantes a la hora de desarrollar un videojuego se pueden desglosar en bastantes apartados pequeños, lo cual nos permite establecer un calendario basado en pequeños objetivos a corto plazo que se basará en la realización de dichas pequeñas tareas (por ejemplo, se puede desglosar el desarrollo del personaje en pequeñas tareas como programación del movimiento, implementación de acciones,

e interacción con el entorno, entre otras). Aparte de estas pequeñas tareas, se pueden constituir apartados más grandes y genéricos que se pueden ordenar según su prioridad a la hora del desarrollo y la implementación de los mismos.

Todas las tareas de desarrollo incluyen una pequeña fase previa de análisis y diseño, seguida de su implementación, y acabando por las pruebas necesarias para comprobar el buen funcionamiento de la característica programada. Además, el orden establecido es importante ya que tareas posteriores generalmente no se pueden llevar a cabo sin completar las anteriores (en algunos casos particulares, algunas tareas no son bloqueantes de tareas posteriores, como el hito inicial).

A continuación se muestra la lista ordenada de dichas tareas:

1. Hito inicial (punto actual)
2. Configuración de las herramientas y el entorno de trabajo: Instalación y configuración del software necesario (Unity, Visual Studio, Git Bash)
3. Estudio de las técnicas de IA y diseño del bot para el juego
4. Diseño e implementación del videojuego en Unity
5. Implementación de la IA
6. Pruebas y balanceo del bot
7. (opcional) Experimentos adicionales (combate por equipos, *bot vs bot*, etc.)
8. Hito final: Redacción de la memoria, anexos, documentación necesaria.

5.2.4 Hito final

En esta última fase del proyecto se redactará y se presentará la memoria del trabajo, además de realizar la lectura del mismo en la fecha establecida. Junto a la memoria se entregará el proyecto de Unity con todo el código fuente, y la documentación adecuada, además de una versión compilada del juego, una pequeña guía de uso para aprender las características básicas del mismo, y todos los documentos y ficheros que sean necesarios para la entrega final.

5.2.5 Duración aproximada

Tarea	Duración aproximada (horas)
Hito inicial	80
Análisis y diseño del proyecto	40
Configuración de herramientas	10
Diseño de la IA	50
Diseño e implementación del juego	150
Implementación de la IA	50
Pruebas y balanceo	50
Experimentos adicionales	40
Hito final	40
Total	**510**

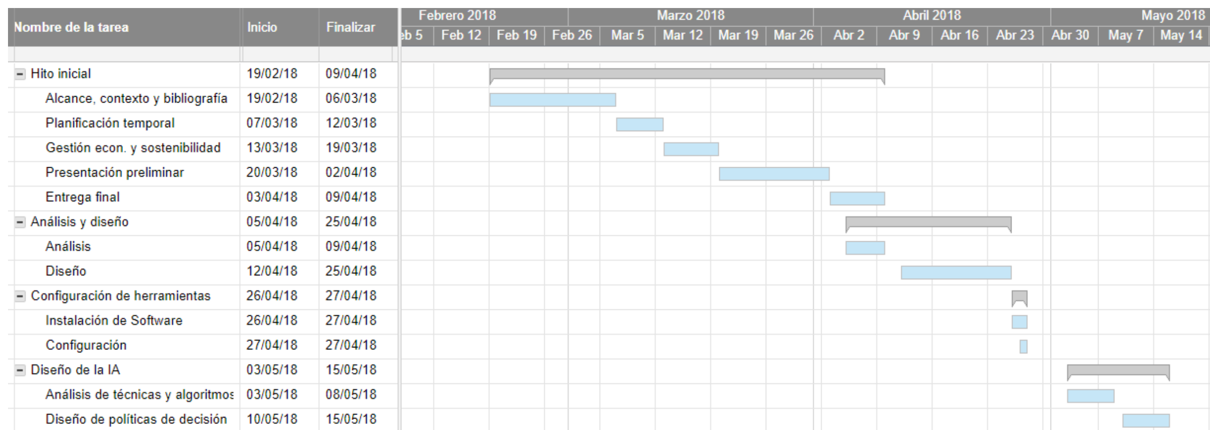
5.2.6 Valoración de alternativas

Dado que hay obstáculos que se presentan durante el desarrollo del proyecto, la planificación puede sufrir modificaciones debido a que algunas tareas pueden requerir más tiempo del esperado debido a errores inesperados que se tienen que arreglar fuera de calendario. Esto se ha podido observar durante el desarrollo de este proyecto.

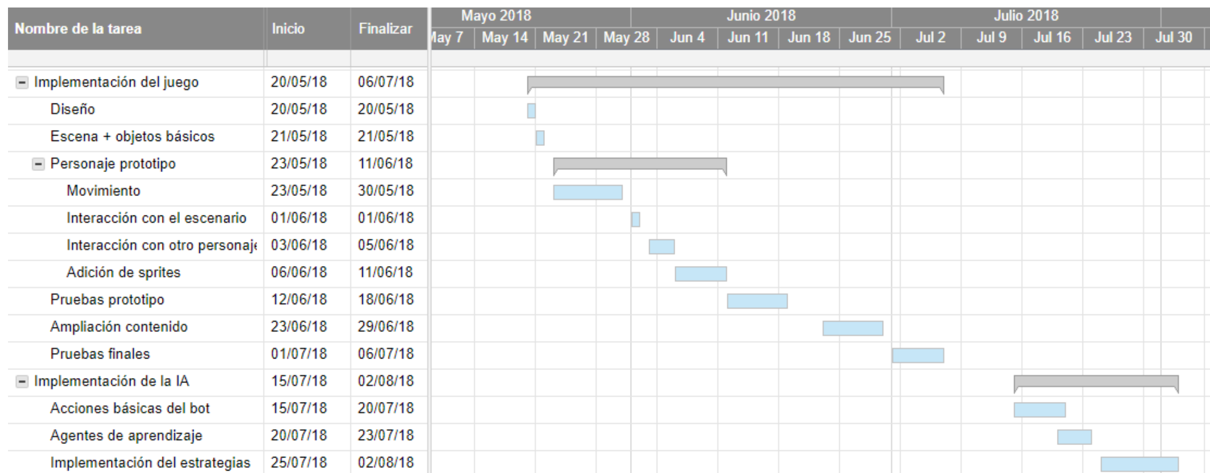
Se procuró seguir la planificación temporal del proyecto, pero finalmente ha sido necesario extender el período de desarrollo de cada una (sin aumentar las horas necesarias para el desarrollo) y se ha mantenido el apartado de experimentos adicionales, que es algo que se ha desarrollado como extra en caso de que haya tiempo de sobras al acabar el proyecto (debido a la extensión, en este caso). También se han dedicado algunas horas extras a algunas de las partes principales del desarrollo.

5.2.7 Diagrama de Gantt

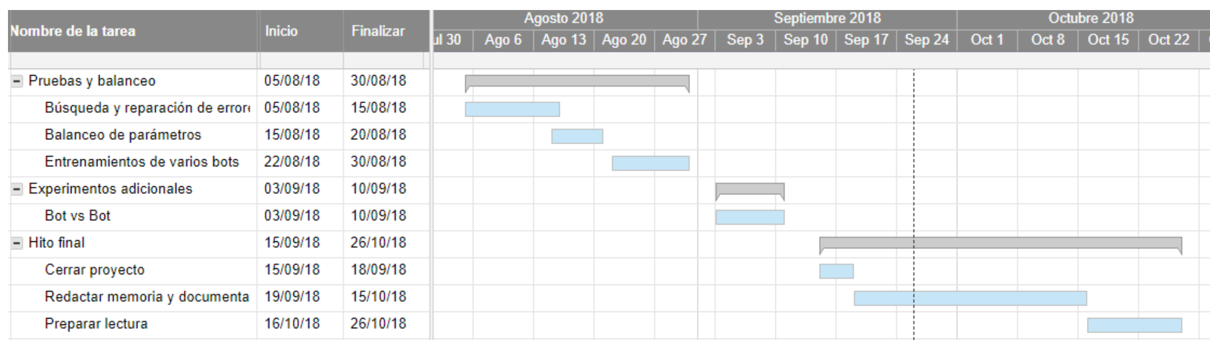
Dividido en tres imágenes para facilitar la visualización. Este diagrama muestra la nueva planificación, obtenida una vez realizada la mayor parte del trabajo, e incluye los nuevos intervalos de tiempo requeridos para cada parte del trabajo.



Nombre de la tarea	Inicio	Finalizar
Hito inicial	19/02/18	09/04/18
Alcance, contexto y bibliografía	19/02/18	06/03/18
Planificación temporal	07/03/18	12/03/18
Gestión econ. y sostenibilidad	13/03/18	19/03/18
Presentación preliminar	20/03/18	02/04/18
Entrega final	03/04/18	09/04/18
Análisis y diseño	05/04/18	25/04/18
Análisis	05/04/18	09/04/18
Diseño	12/04/18	25/04/18
Configuración de herramientas	26/04/18	27/04/18
Instalación de Software	26/04/18	27/04/18
Configuración	27/04/18	27/04/18
Diseño de la IA	03/05/18	15/05/18
Análisis de técnicas y algoritmos	03/05/18	08/05/18
Diseño de políticas de decisión	10/05/18	15/05/18



Nombre de la tarea	Inicio	Finalizar
Implementación del juego	20/05/18	06/07/18
Diseño	20/05/18	20/05/18
Escena + objetos básicos	21/05/18	21/05/18
Personaje prototipo	23/05/18	11/06/18
Movimiento	23/05/18	30/05/18
Interacción con el escenario	01/06/18	01/06/18
Interacción con otro personaje	03/06/18	05/06/18
Adición de sprites	06/06/18	11/06/18
Pruebas prototipo	12/06/18	18/06/18
Ampliación contenido	23/06/18	29/06/18
Pruebas finales	01/07/18	06/07/18
Implementación de la IA	15/07/18	02/08/18
Acciones básicas del bot	15/07/18	20/07/18
Agentes de aprendizaje	20/07/18	23/07/18
Implementación del estrategias	25/07/18	02/08/18



Nombre de la tarea	Inicio	Finalizar
Pruebas y balanceo	05/08/18	30/08/18
Búsqueda y reparación de errore	05/08/18	15/08/18
Balanceo de parámetros	15/08/18	20/08/18
Entrenamientos de varios bots	22/08/18	30/08/18
Experimentos adicionales	03/09/18	10/09/18
Bot vs Bot	03/09/18	10/09/18
Hito final	15/09/18	26/10/18
Cerrar proyecto	15/09/18	18/09/18
Redactar memoria y documenta	19/09/18	15/10/18
Preparar lectura	16/10/18	26/10/18

5.2.8 Recursos para el desarrollo del proyecto

Para llevar a cabo del proyecto bajo esta planificación, se han necesitado los siguientes recursos, tanto *hardware* como *software*.

5.2.8.1 Hardware

- Ordenador de sobremesa estándar.
- Portátil para las pruebas del juego.
- Periféricos necesarios
    - Dispositivos de juego (Mandos)

5.2.8.2 Software

- Windows 10
- Unity
- Visual Studio
- Unity cloud storage
- Google Docs
- GIMP

Este listado de recursos ha cambiado durante el desarrollo del proyecto respecto a la versión inicial, dadas las necesidades que han ido surgiendo.

5.3 Gestión económica

5.3.1 Recursos humanos

Este proyecto se ha desarrollado por una sola persona, que ha tenido que realizar el trabajo de todos los roles: Jefe de proyecto, diseñador, programador y tester. Se deben tener en cuenta las horas de trabajo asignadas a cada rol de cara al proyecto, a la hora de analizar el coste económico referente a recursos humanos.

Rol	Coste/hora	Horas	Coste total
Jefe de proyecto	50 €/h	120 h	6.000 €
Diseñador	35 €/h	110 h	3.850 €
Programador	30 €/h	240 h	7.200 €
Tester	20 €/h	40 h	800 €
Total		510 h	17.850 €

5.3.2 Hardware

Es necesario cubrir los costes de hardware para el desarrollo del proyecto, en este caso, como se comentó anteriormente, sería un pc de sobremesa estándar, un portátil, y un par de dispositivos de juego. Consideramos que su reparación entra dentro del precio de coste dado que contamos con garantía extendida en todos los dispositivos.

Producto	Precio/u	Unidades	Vida útil	Amortización
PC sobremesa personalizado	1.199 €	1	5 años	159,86 €
Asus GL753VD-GC009	849,01 €	1	5 años	113,20 €
Mando XBox 360 para PC	39,99 €	2	2 años	26,66 €
Total	2.127,99 €			299,72 €

5.3.3 Software

Igual que en el apartado anterior, también debemos contar los costes relacionados con el software utilizado.

Producto	Precio/u	Unidades	Vida útil	Amortización
Windows 10	145 €	1	5 años	19,30 €
Unity	0 €	1	-	-
Visual Studio Community 2017	0 €	1	-	-
Git Bash	0 €	1	-	-

Google Docs	0 €	1	-	-
GIMP	0 €	1	-	-
Total	145 €			19,30 €

5.3.4 Licencias

También hay que tener en cuenta los gastos en licencias de desarrollo del software que se ha utilizado, o licencias de publicación en caso de que se publique en alguna plataforma de pago.

Producto	Precio/u	Unidades	Vida útil	Amortización
Unity Pro	100 €/mes	1	8 meses	800 €
Total	400 €			800 €

5.3.5 Gastos indirectos

Finalmente debemos considerar los gastos extra que no están directamente relacionados con el proyecto, pero que son igual de necesarios.

Producto	Precio	Unidades	Coste aproximado
Electricidad	0,12 €/kWh	950 kWh	114 €
Internet	26,90 €/mes	9 meses	242,10 €
Pack material oficina	45,99 €	1	45,99 €
Total			402,09 €

5.3.6 Presupuesto total

Así pues, si juntamos todos los gastos establecidos en los apartados anteriores, podemos ver el coste total estimado del proyecto.

Concepto	Coste aproximado
Recursos humanos	17.850 €
Hardware	299,72 €

Software	19,30 €
Licencias	800 €
Gastos indirectos	402,09 €
Total	19.371,11 €

5.4 Sostenibilidad

5.4.1 Dimensión ambiental

Desde el punto de vista ambiental, este proyecto no tiene apenas impacto. Dada la naturaleza software del mismo, lo único que se requiere es un ordenador para acceder al contenido y jugar al juego.

Por lo tanto, el único aspecto relacionado con la cuestión ambiental va dirigida al hardware utilizado para el desarrollo del proyecto. Dada su corta duración, el material obtenido para ello (principalmente todos los componentes hardware) prácticamente no va a ser amortizado, con lo cual es importante considerar volver a utilizarlo en futuros proyectos para aprovechar al máximo su vida útil.

5.4.2 Dimensión social

A nivel social, el impacto que pueda tener el proyecto se debe medir centrándose principalmente en el público al que va dirigido: los jugadores de videojuegos. Al tratarse de entretenimiento, no existe una necesidad real del proyecto, pero al fin y al cabo es igual que todos los demás ejemplos de entretenimiento: no es algo necesario para la vida, pero le da sentido más allá del aspecto biológico de la misma. Por ello debemos siempre intentar ir más allá en el desarrollo de nuevos métodos de entretener a las personas.

Es cierto que actualmente las empresas y los desarrolladores independientes intentan buscar soluciones novedosas a la cuestión del entretenimiento en forma de videojuegos. Hay que buscar el punto medio entre originalidad e historia: Intentar buscar algo demasiado novedoso puede impactar negativamente en el hecho de

que no llame la atención o no le guste a la gente, mientras que hacer una y otra vez lo mismo hará que se vuelva repetitivo o aburrido. Es por ello que hay que buscar evolucionar poco a poco, y en cierto modo es lo que se intenta con este proyecto, aunque no sea completamente innovador en el mundo de los videojuegos.

Personalmente, este proyecto ha sido más bien un desafío contra mí mismo, tanto como desarrollador de videojuegos, como jugador. Me ha permitido ir más allá en un mundo que me atrae desde hace años, al mismo tiempo que me ha otorgado la posibilidad de construir algo que no es muy común actualmente, y que puede abrir nuevas vías en el mundo de los videojuegos en general.

5.4.3 Matriz de sostenibilidad

Tras el desarrollo del proyecto, se puede extraer la siguiente matriz de sostenibilidad a partir de los apartados anteriores y el desarrollo del juego.

Sostenibilidad	Económica	Social	Ambiental
Hito inicial	6	8	8
Hito intermedio	7	8	8
Hito final	7	8	8

Económicamente ha mejorado dado que durante el desarrollo se ha podido mejorar el uso de los medios y se ha amortizado más el equipo. En los demás aspectos se ha mantenido igual que lo esperado al inicio del desarrollo.

6. Conclusiones

Llegados a este punto, podemos confirmar que el mundo de las IAs en videojuegos todavía tiene un gran panorama por explorar. Desde IAs estáticas pre-procesadas como las que usan la mayoría de videojuegos de hoy en día, hasta IAs dinámicas que van aprendiendo a medida que avanza el juego, como la que se ha implementado en este proyecto.

Es un tema que aún ofrece horizontes que descubrir y alcanzar, tanto desde un punto de vista teórico (búsqueda de nuevos algoritmos de aprendizaje por refuerzo, o variantes de los ya existentes que ofrezcan mejores resultados), como en la práctica (implementación de nuevas estrategias y políticas de toma de decisión, e incluso nuevos sistemas dedicados al aprendizaje por refuerzo y a la IA en general).

Centrándonos en el caso particular del juego que se ha implementado en este proyecto, podemos afirmar que se han alcanzado los objetivos establecidos al inicio del mismo:

- Hemos creado un videojuego de lucha 2D
- Hemos implementado un bot manejado por un agente de aprendizaje que va aprendiendo a medida que juega
- Hemos comprobado el buen funcionamiento del agente mediante el diseño e implementación de distintas estrategias de aprendizaje

Además, durante el desarrollo del proyecto se han encontrado obstáculos que se han tenido que superar con el fin de poder seguir adelante, y al mismo tiempo se han abierto nuevas metas de cara al futuro. En conjunto, el resultado obtenido es satisfactorio, y anima a seguir investigando este campo en el futuro.

6.1 Posibles ampliaciones

Este trabajo está acabado, pero tanto el tema de la IA como el desarrollo del propio videojuego tienen abiertos amplios horizontes que se pueden explorar en un futuro:

- Ampliación del contenido del juego: Añadir nuevos personajes, escenarios y modos de juego, para incrementar el nivel de diversión que ofrece.
- Nuevas características que podrían hacer interesante el juego, como la posibilidad de enfrentar entre sí bots entrenados por distintos jugadores, o incluso torneos de agentes.
- Más tipos de pruebas y experimentos usando el sistema implementado de agentes de aprendizaje; por ejemplo, entrenar bots luchando entre ellos durante horas, o enfrentar agentes que sigan distintas estrategias.
- Exploración de otros algoritmos de aprendizaje por refuerzo: en este proyecto hemos aplicado PPO, pero hay muchas otras opciones que se podrían explorar e implementar, y observar qué resultados ofrecen en comparación con los realizados hasta ahora.

Bibliografía

[1] http://aigamedev.com/open/review/top-ai-games/ - Top 10 most influential AI games

[2] http://gameaibook.org/ - Artificial Intelligence and Games

[3] https://unity3d.com/machine-learning - Unity Machine Learning Agents

[4] https://medium.com/@pavelkordik/reinforcement-learning-the-hardest-part-of-machine-learning-b667a22995ca - Reinforcement Learning: Artificial Intelligence in Game Playing

[5] http://cs231n.github.io/neural-networks-1/ - Convolutional Neural Networks for Visual Recognition

[6] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html - Introduction to A*

[7] https://unity3d.com/ - Unity 3D

[8] https://www.unrealengine.com/ - Unreal Engine

[9] https://pegi.info/ - PEGI (Pan European Game Information)

[10] http://www.esrb.org/ - ESRB (Entertainment Software Rating Board)

[11] https://hisouten.koumakan.jp/ - Touhou Hisoutensoku wiki

[12] https://www.spriters-resource.com/pc_computer/touhou123/?source=genre - Web de descarga de *spritesheets*

[13] https://towardsdatascience.com/ - Towards Data Science - Machine Learning Blog.